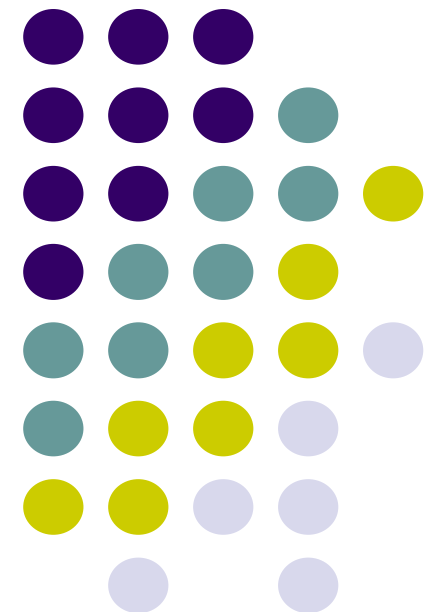
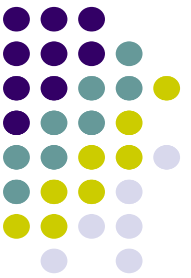
Methods for Software Specification

Prof. Ing. Ivo Vondrak, CSc.
Dept. of Computer Science
Technical University of Ostrava
ivo.vondrak@vsb.cz
http://vondrak.cs.vsb.cz

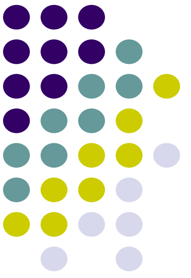




References

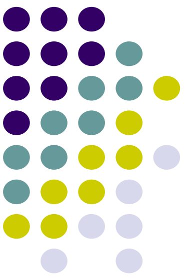
- Rumbaugh, James et al. Object-Oriented Modeling and Design, Prentice Hall Inc. 1991
- Booch, Grady: Object-Oriented Analysis and Design, The Benjamin/Cummings Publishing Company, Inc. 1994
- Jacobson, I., Christerson, M., Jonsson, P., Overgaard, G.: Object Oriented Software Engineering, A Use Case Driven Approach, Addison-Wesley, 1994
- Gamma, E., Helm,R., Johnson,R., Vlissides,J. Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994
- Booch, G., Jacobson, I., Rumbaugh, J.: The Unified Modeling Language User Guide, Addison Wesley Longman, Inc., 1999
- Schmuller, J.: Teaching Yourself UML in 24 Hours, Sams, 1999
- OMG Unified Modeling Language Specification, version 1.5, http://www.uml.org, 2003
- Warmer, J., Kleppe, A.: The Object Constraint Language Second Edition, Getting Your Models Ready for MDA, Addison-Wesley, 2003

Contents



- Introduction
- UML Language
- Formal Methods of Specification
- Design Patterns

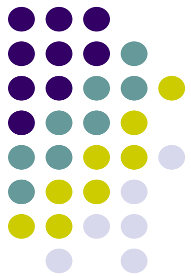
Definition of OOM



- **Method** is well-considered (sophisticated) system of doing or arranging something.
- **Architecture** is the organizational structure and associated behavior of a system.
- **Object** is an entity with a well-defined boundary and identity that encapsulates state and behavior. An object is an instance of a class.
- **Object-oriented architecture** is the structure of connected objects that define resulting behavior of the system through their communication (interactions).

Object-oriented method is sophisticated system of doing software based on object-oriented architecture.

Three Sources of UML



Booch Method
...
G. Booch

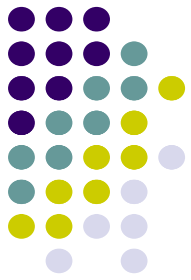
OOSE
Object-oriented Software Engineering
I. Jacobson

OMT
Object Modeling Technique
J. Rumbaugh

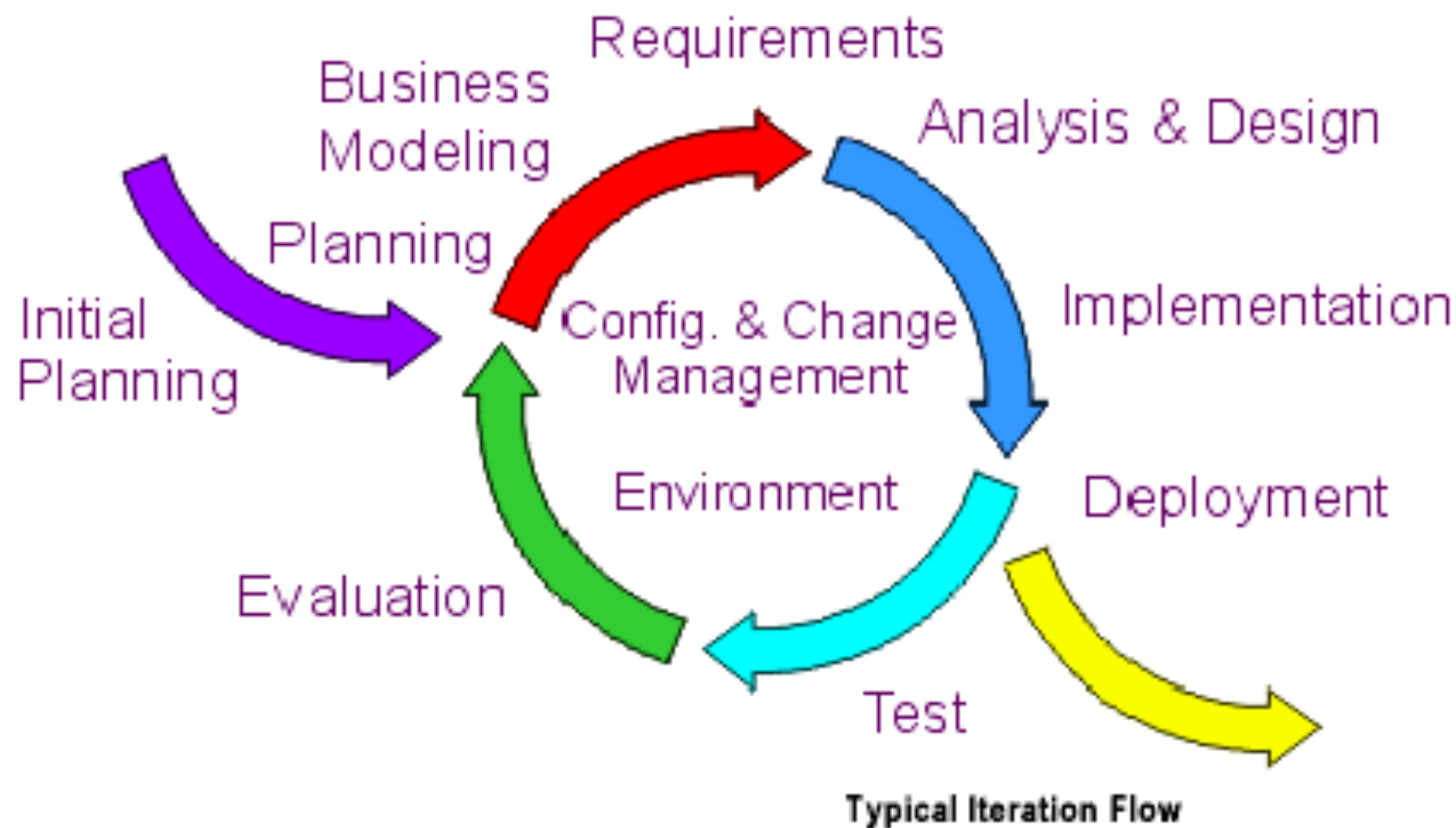
UML
Unified Modeling Language
Rational

Standardized by OMG

Iterative Software development

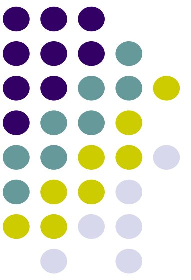


An iterative approach is required that allows an increasing understanding of the problem through successive refinements, and to incrementally grow an effective solution over multiple iterations.



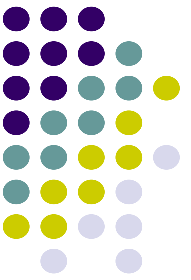
Typical Iteration Flow

Core Engineering Workflows



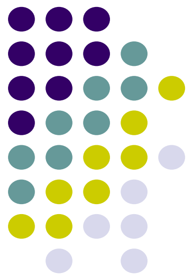
- **Business modeling** describes the structure and dynamics of the organization
- **Requirement** describe the use case-based method for eliciting requirements
- **Analysis and design** describe the multiple architectural views
- **Implementation** takes into account sw development, unit test, and integration
- **Test** describes test cases and procedures
- **Deployment** covers the deliverable system configuration

The Unified Modeling Language



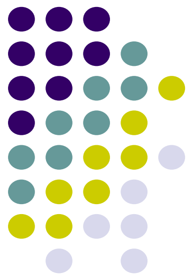
- The Unified Modeling Language (UML) is a standard language for writing software blueprints.
- The UML may be used to **visualize**, **specify**, **construct** and document the artifacts of a software-intensive system.
  - **Visualizing** means graphical language
  - **Specifying** means building precise, unambiguous, and complete models
  - **Constructing** means that models can be directly connected to a variety of programming languages

Language Architecture



Layer	Description	Example
Meta-metamodel	The infrastructure for a metamodeling architecture. Defines the language for specifying metamodels.	MetaClass, MetaAttribute, MetaOperation
Metamodel	An instance of a meta-metamodel. Defines the language for specifying a model.	Class, Attribute, Operation, Component
Model	An instance of a metamodel. Defines a language to describe an information domain.	StockShare, askPrice, sellLimitOrder, StockQuoteServer
User Objects (User Data)	An instance of a model. Defines a specific information domain.	<Acme_SW_Share_98789>, 654.56, sell_limit_order, <Stock_Quote_Svr_32123>

Four-layer Metamodel Architecture



UML Diagrams

- Use Case diagram
- Class diagram
- Behavior diagrams:
  - Statechart diagram
  - Activity diagram
- Interaction diagrams:
  - Sequence diagram
  - Collaboration diagram
- Implementation diagrams:
  - Component diagram
  - Deployment diagram

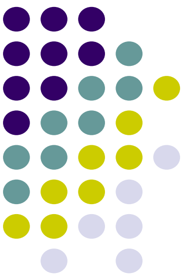
Functional View

Logical View

Behavioral View

Implementation View

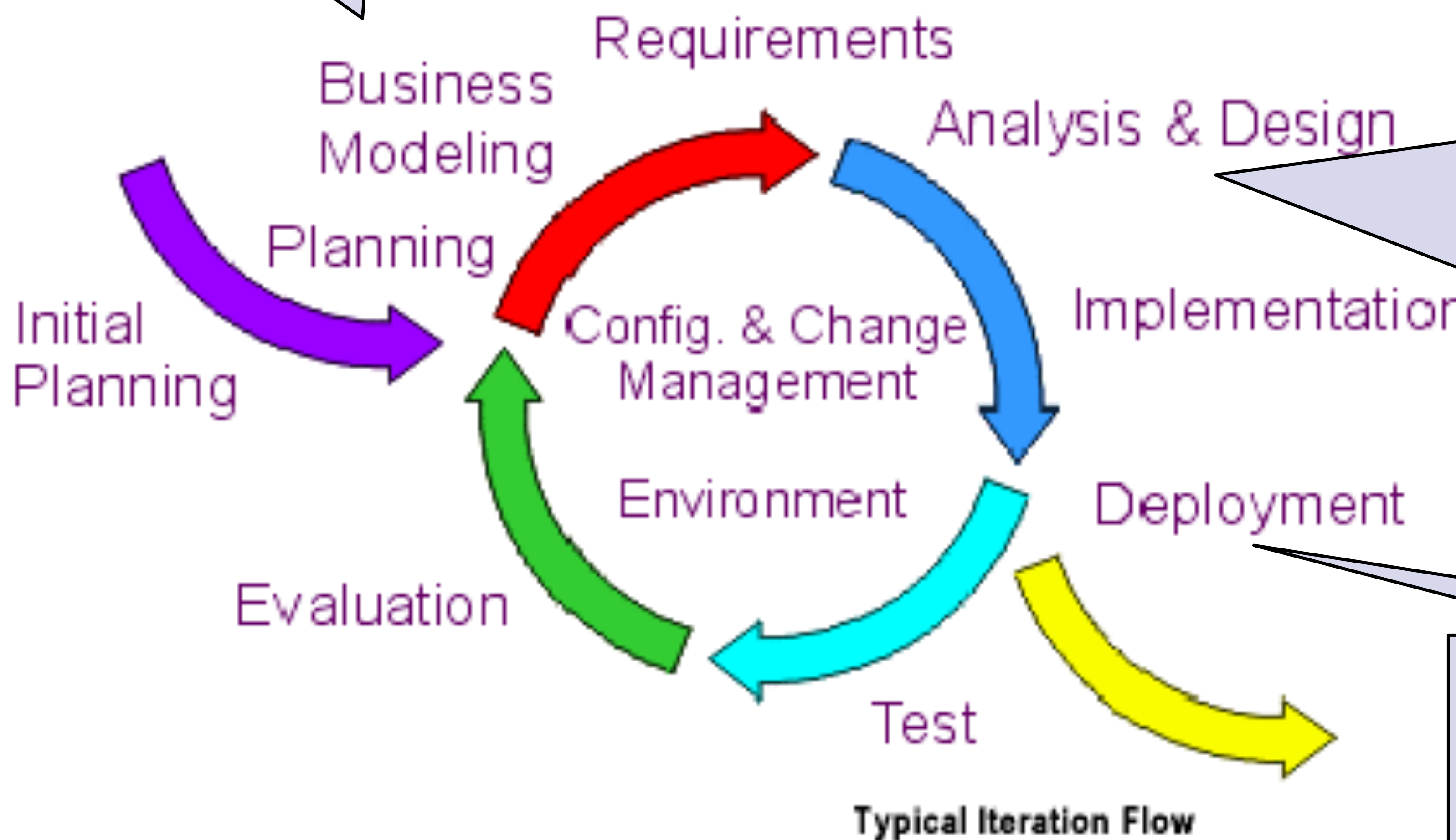
Software Lifecycle and UML Diagams



Activity diagrams
Class diagrams

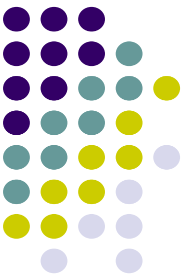
Use Case diagrams
Sequence diagrams

Class diagams
Sequence diagrams
Collaboration diagrams
Statechart diagrams
Deployment diagrams

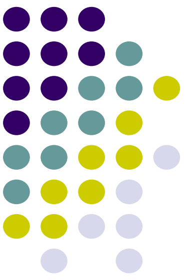


Component diagrams
Deployment diagrams

Exercises



- What is the object-oriented methods about? What is the **definition of the object-oriented method**?
- **What diagrams are used to specify each workflow** of the software development iteration?
- What is the UML language architecture. **Describe four-layer metamodel architecture**!

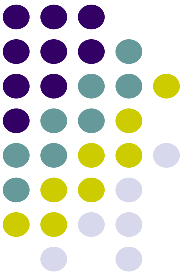


Functional View of the System

Use case is the specification of a sequence of actions, including variants, that a system (or other entity) can perform, interacting with actors of the system.

- **Use case diagram** shows the relationships among actors and use cases within a system.
- **Actor** is coherent set of roles that users of use cases play when interacting with these use cases. An actor has one role for each use case with which it communicates.

Use Case Diagram



System border

e-Shop

Customer

Order Processing

Salesman

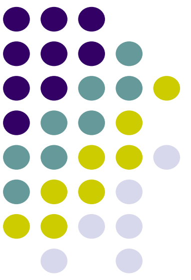
Actor

Product delivery

Use Case

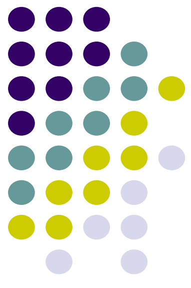
Dispatch worker

Relationship between Use Cases



- **Extend** is a relationship from an extension use case to a base use case, specifying how the behavior defined for the extension use case extends the behavior defined for the base use case. The base use case does not depend on performing the behavior of the extension use case.
- **Include** is a relationship from a base use case to an inclusion use case, specifying how the behavior for the base use case contains the behavior of the inclusion use case. The base use case depends on performing the behavior of the inclusion use case.
- **Generalization** is a relationship between a more general use case and its more specific version.

Use Case Relationships Example



Customer data entry

Payment checking

«include»

«include»

Include relationship

Salesman

Order processing

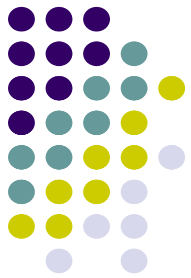
Extend relationship

«extends»

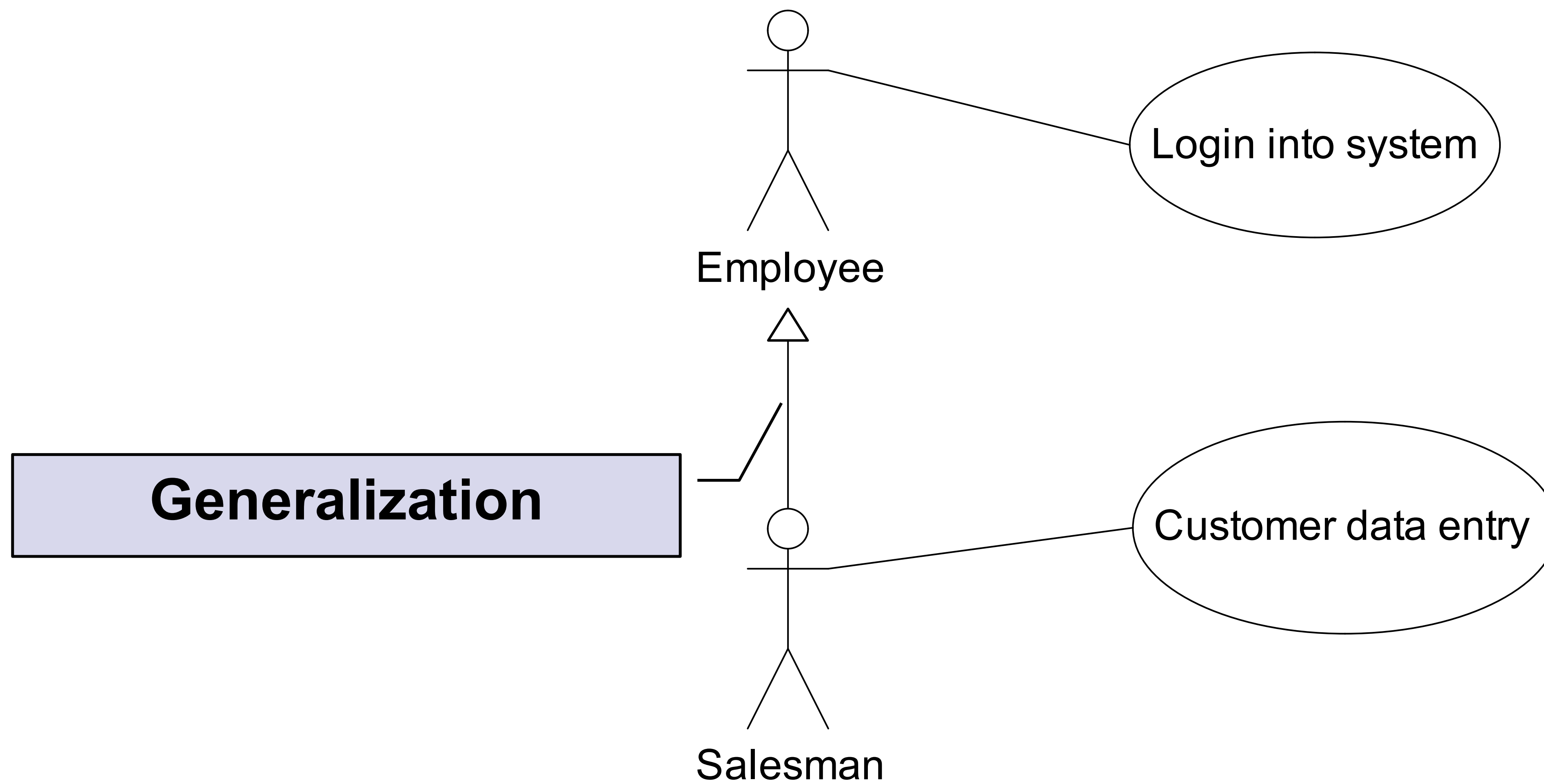
Generalization

Catalog offering

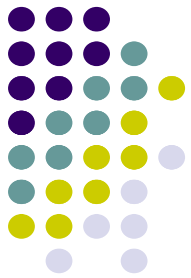
Express order processing



Relationships between Actors

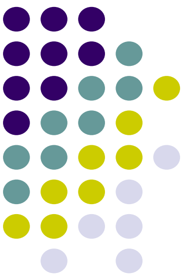


Exercises



- What entities are used by use case diagrams?
- **List and describe all kind of relationships** used by use case diagram specification!
- **Specify all use cases employed by the customer of eShop**!

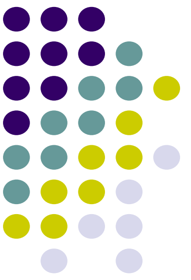
Logical View of the System



Static structure of the system specifies entities, their structure and relationships among them. For that purpose class and object diagrams are used.

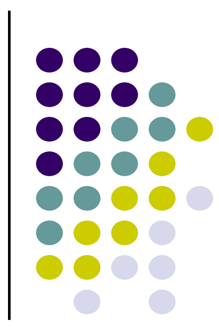
- **Class diagram** shows a collection of declarative (static) model elements, such as classes, types, and their contents and relationships.
- **Object diagram** encompasses objects and their relationships **at a point in time**. An object diagram may be considered a special case of a class diagram or a collaboration diagram.

Class and Type of Object

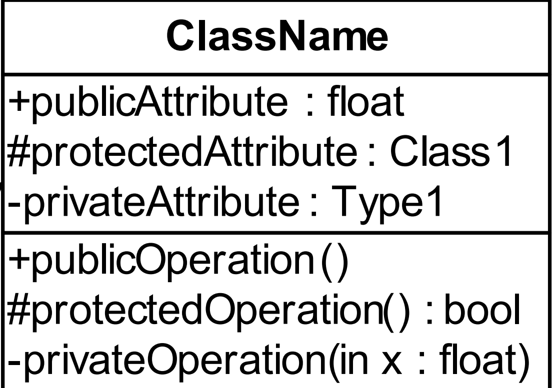


- **Object** is an identifiable individual entity with given identity (a uniqueness which distinguishes it from all other objects) and behavior (services it provides in interactions with other objects) .
- **Class** is a description of a set of objects that share the same attributes, operations, methods, relationships, and semantics. A class may use a set of interfaces to specify collections of operations it provides to its environment.
- **Type** specifies a domain of objects together with the operations applicable to the objects, without defining the physical implementation of those objects. Although an object may have at most one implementation class, it may conform to multiple different types.

Classes, Types and Objects

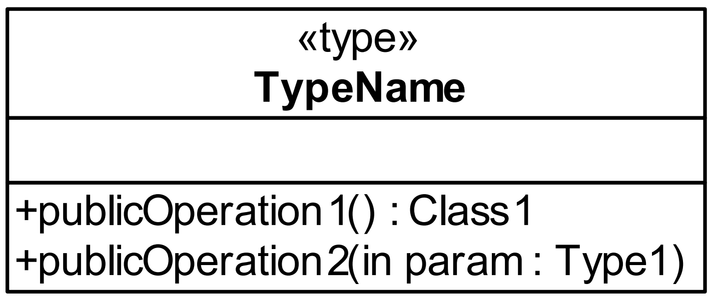


Class

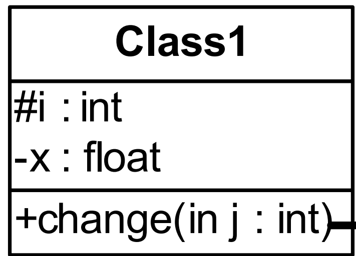


Attributes

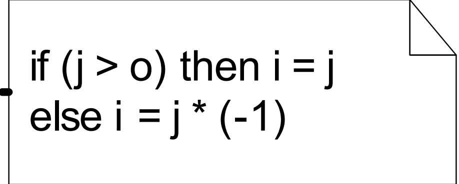
Type



Operations



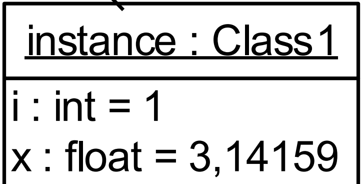
Note

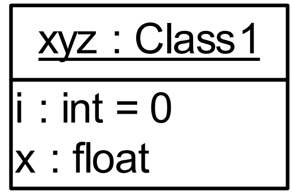


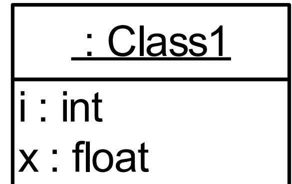
Classes and types

Class instances

Object name

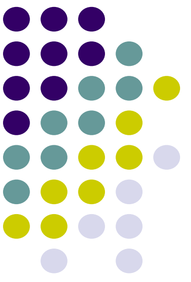






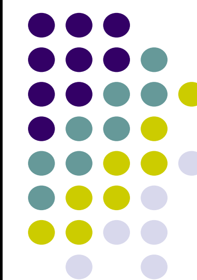
Object

Implementation Class and Interface



- **Implementation class** is said to realize a type if it provides all of the operations defined for the type with the same behavior as specified for the type's operations.
- **Interface** is a named set of operations that characterize the behavior of an element.

Implementation Class Example



«type» Iterator
+next() : Object *+isEmpty() : bool*

Class Catalog implements type Iterator

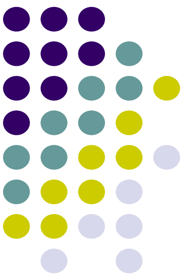
Catalog
-items[] : Object
+add(in o : Object) +remove(in o : Object) +itemAtPosition(in i : int) : Object +next() : Object +isEmpty() : bool

Iterator

Catalog
+add(in o : Object) +remove(in o : Object) +itemAtPosition(in i : int) : Object +next() : Object +isEmpty() : bool

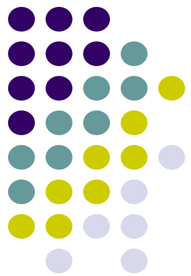
Interface Iterator

Relationships Among Objects and Classes



- **Association** describes a group of links with common structure and common semantics (a Person works-for a Company). An association a bi-directional connection between classes that describes a set of potential links in the same way that a class describes a set of potential objects.
- **Aggregation** is the “part-whole” or “a-part-of” relationship in which objects representing the components of something are associated with an object representing the entire assembly.
- **Dependency** is a weaker form of relationship showing a relationship between a client and supplier.
- **Generalization** is the taxonomic relationship between a more general element (the parent) and a more specific element (the child) that is fully consistent with the first element and that adds additional information.

Association



Role of Object

Association

Employee
-ssn -position
+process() : Order +ship() : Order +checkPayment()

+salesman

is processing

Order
-date -price

Association Name

1..*

ships

-{ordered}

1..*

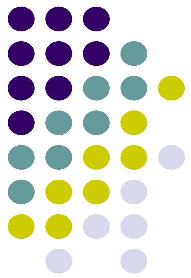
1

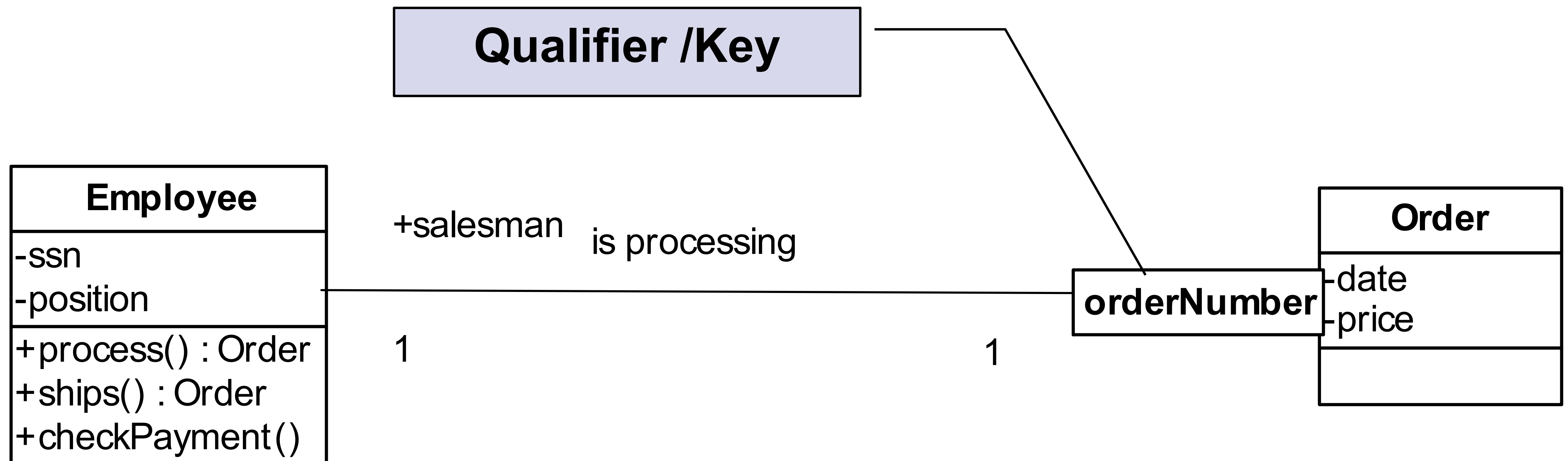
+dispatcher

Ordering

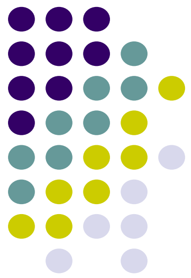
Multiplicity

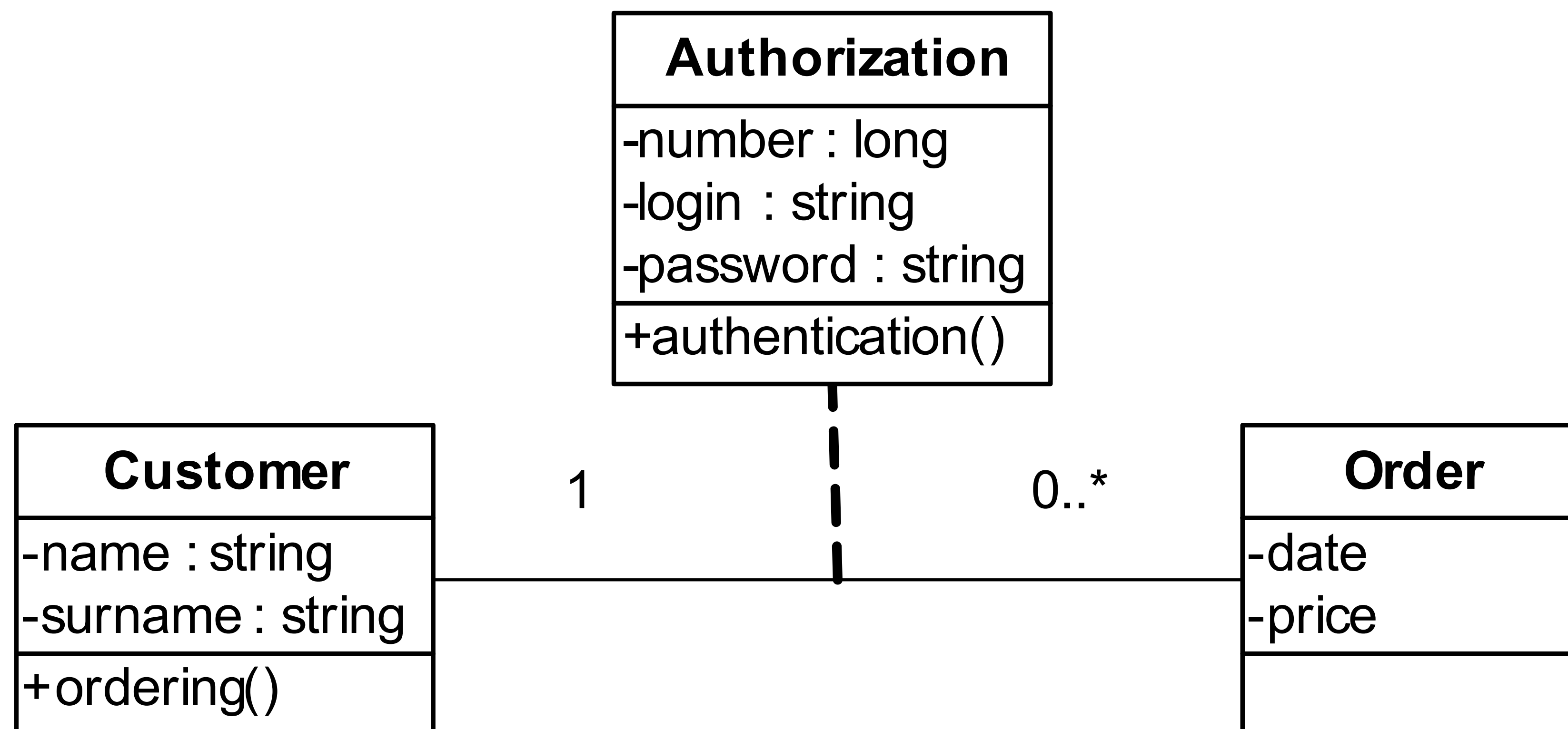
Qualified Association



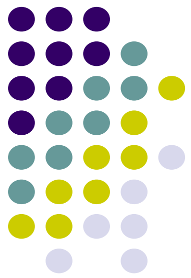


Association Class





Aggregation and Composition



Order
-date -price

Catalog
+add(in o : Object) +remove(in o : Object) +itemAtPosition(in i : int) : Object +next() : Object +isEmpty() : bool

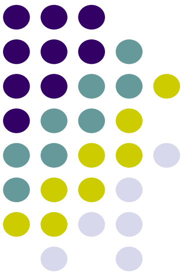
Shared aggregation; the part may be contained in other aggregates.

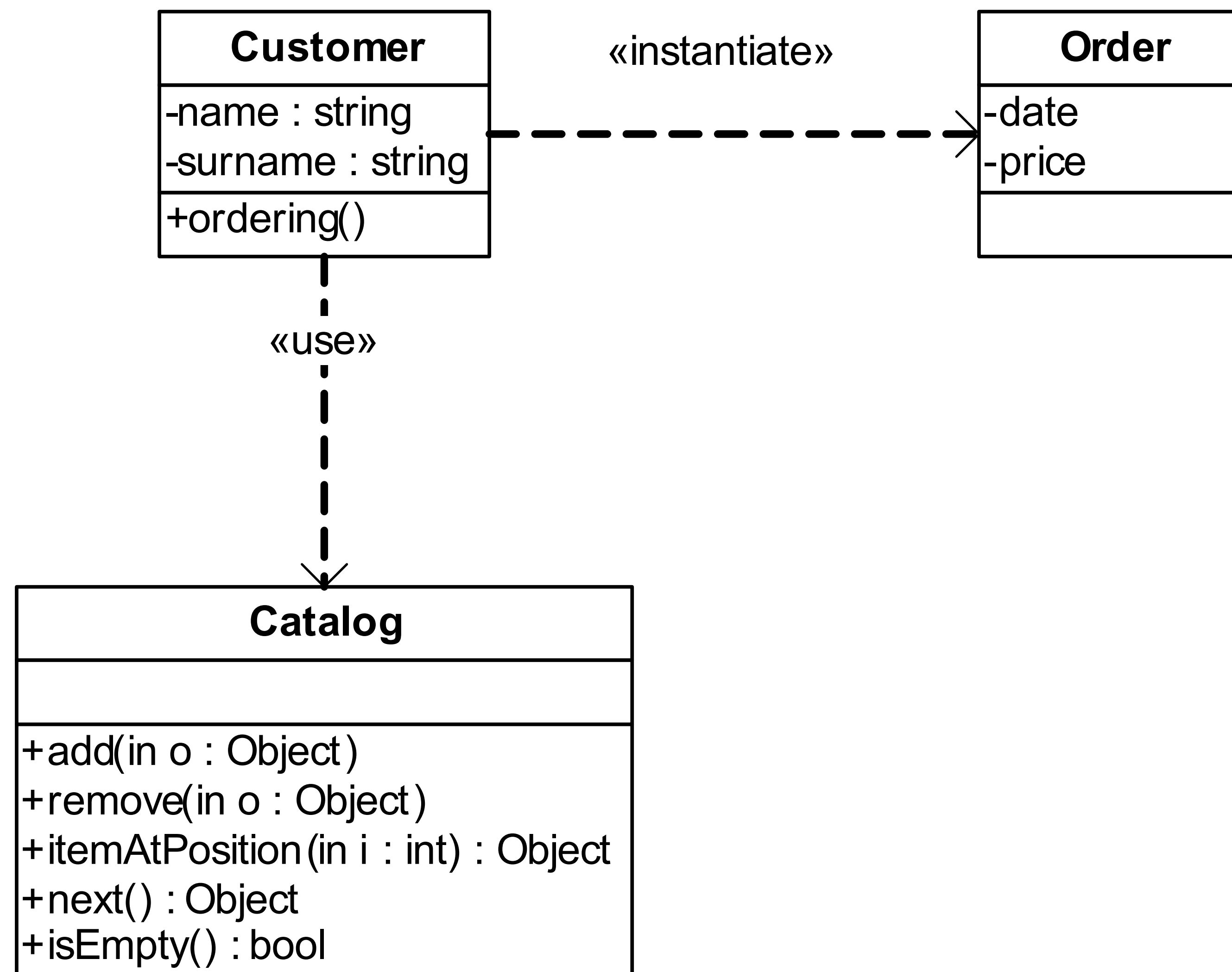
Composition; the part is strongly owned by the composite and may not be part of any other

Zboží
-dodavatel
+sleva()

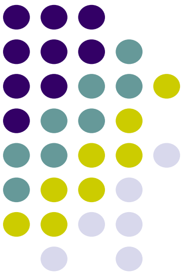
Položka
-název : string -cena : float

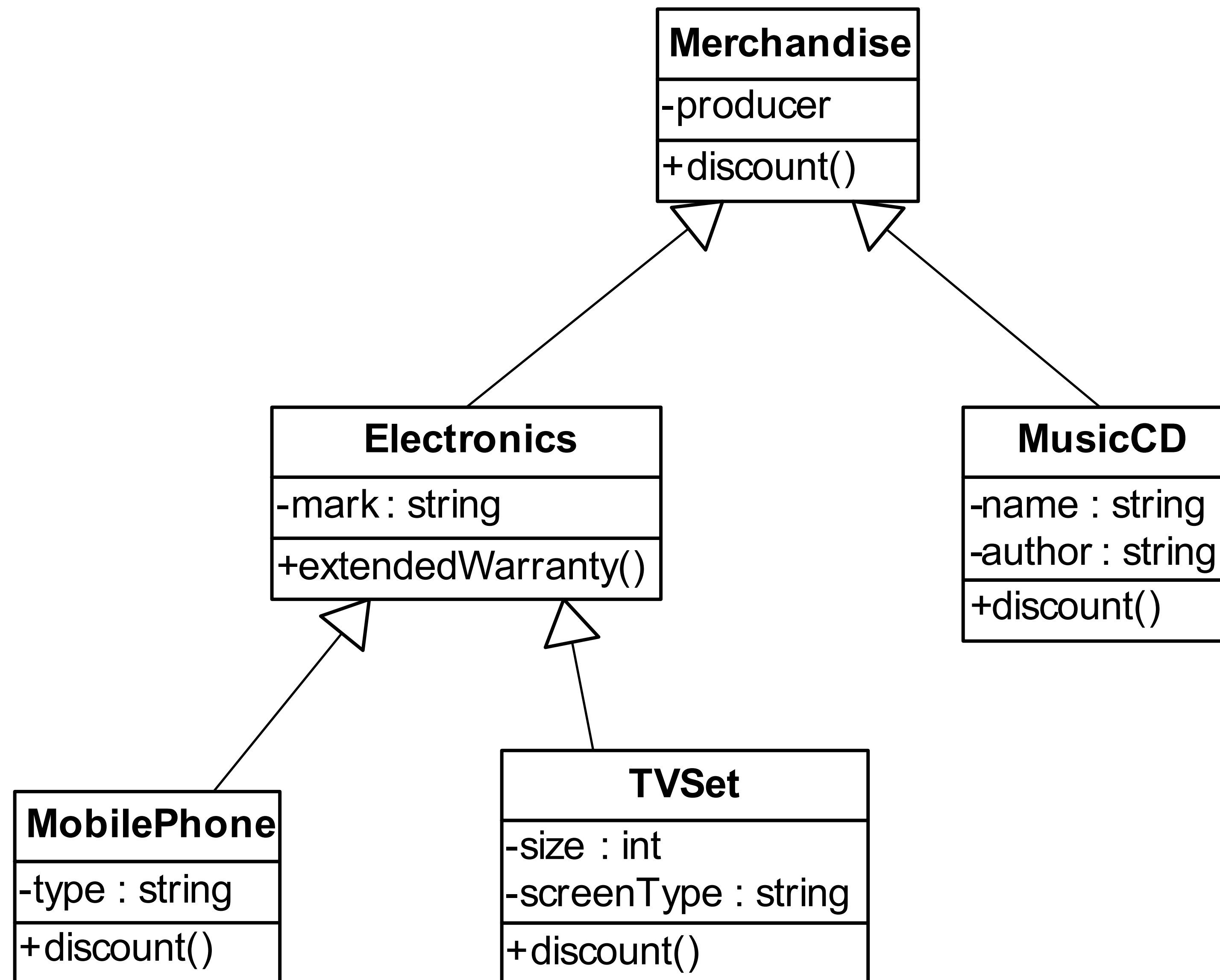
Dependency

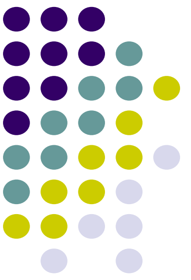




Generalization



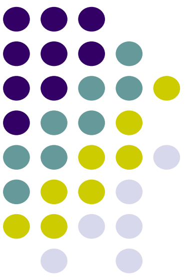




Exercises

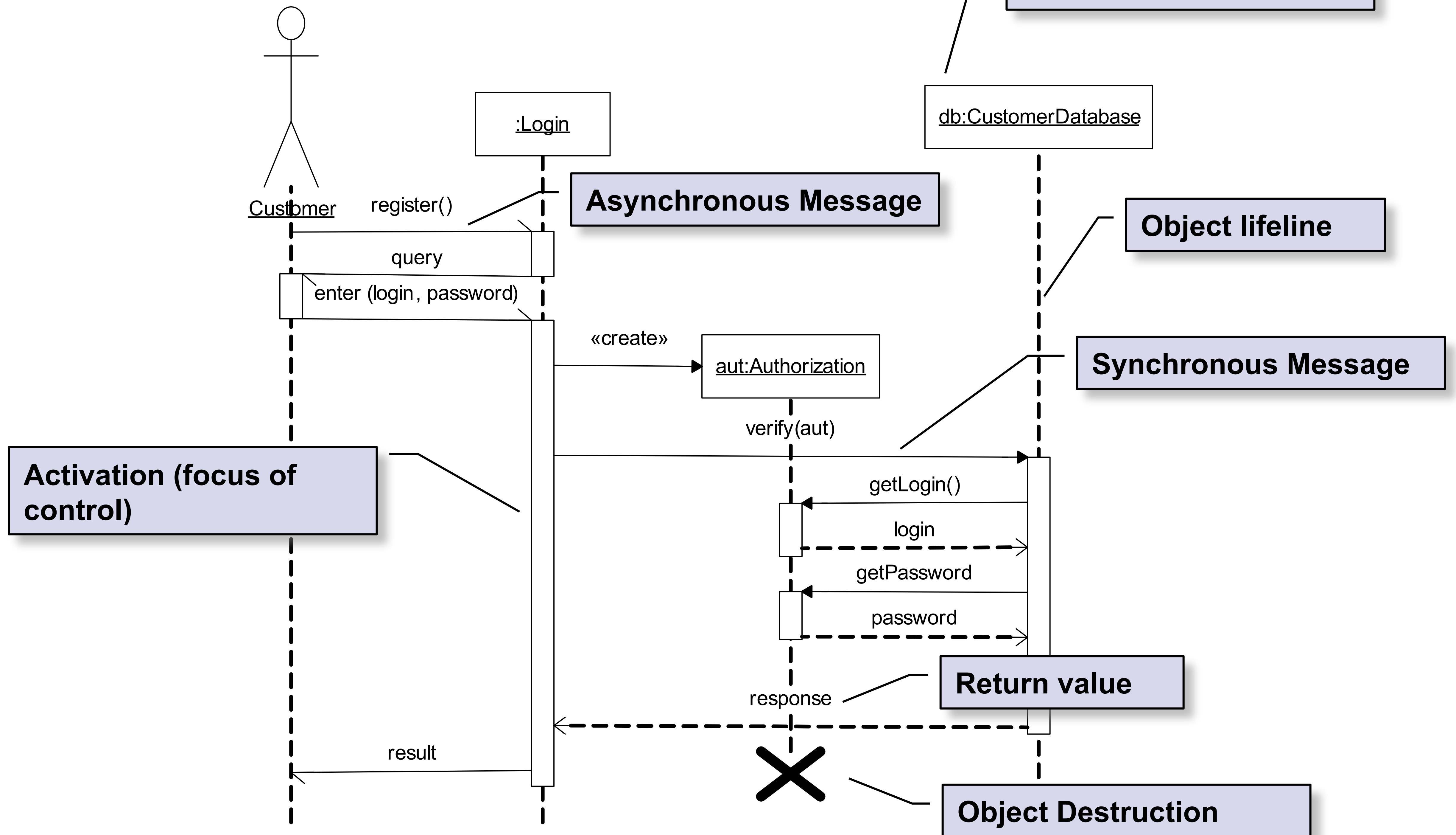
- Define what is the object, class and type!
- What kind of relationships can be defined among classes?
- Build the **class diagram describing academic community**. Academic community consists of educators and students. Educators are assistants, assistant professors, associate professors and professors. Educators play roles of supervisors to other educators that play roles of subordinated.

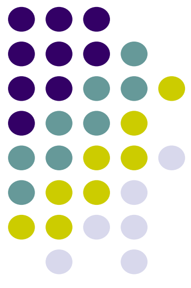
Behavioral View of the System



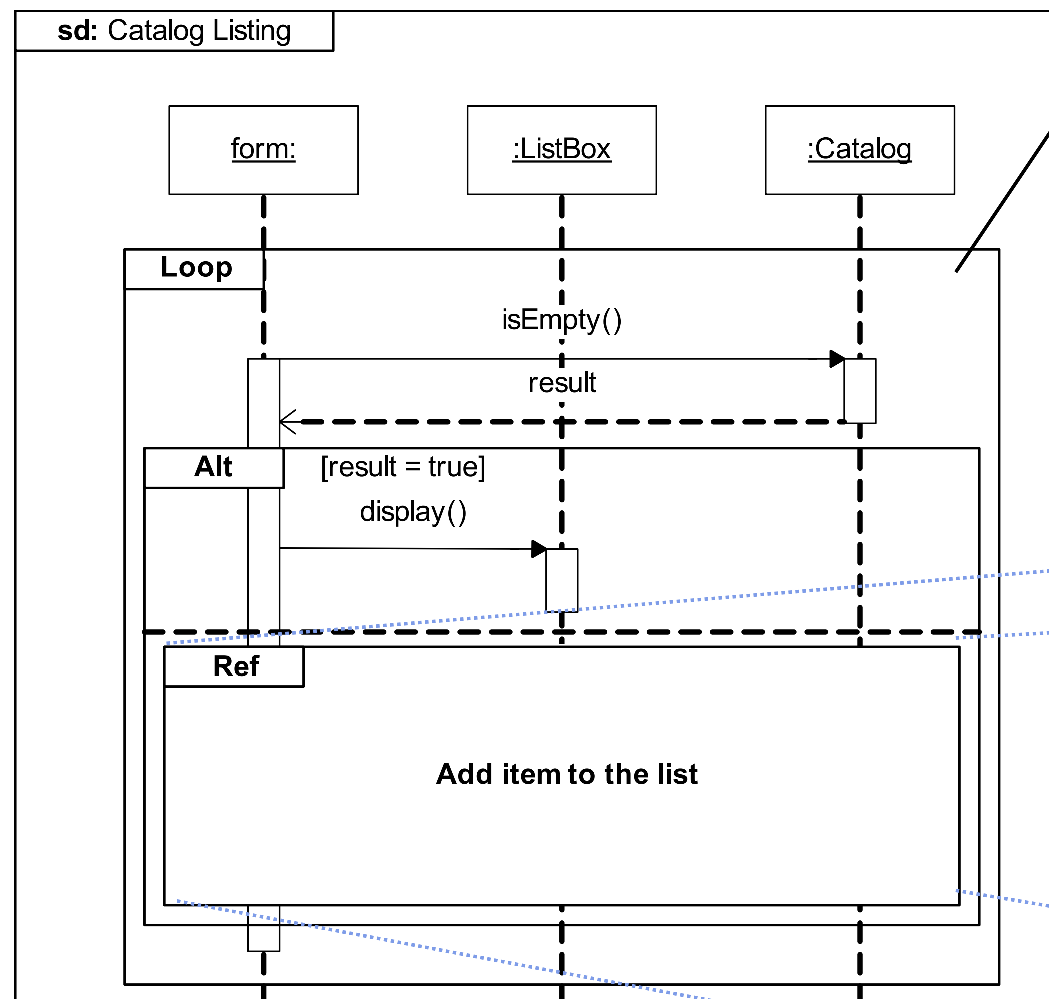
- A set of interconnected objects constitutes the system
- Interactions between objects result in:
  - Collective behaviors being exercised
  - Changes in the logical configurations and states of the objects and system

Sequence Diagram



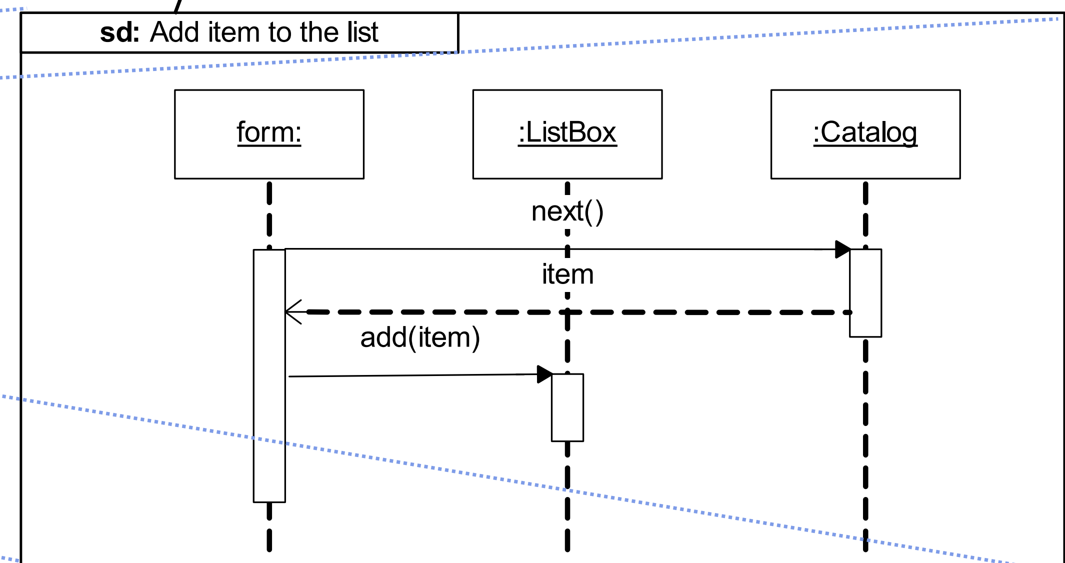


Structured Sequence Diagram

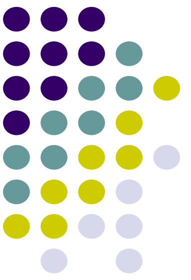


Loop with embedded alternative (**Alt**) and reference to another sequence diagram (**Ref**).

Elaborated Diagram

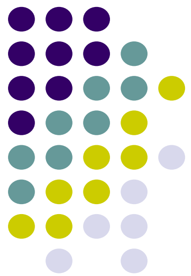


The Dynamic Behavior of an Object



- A **state transition** (statechart) diagram shows
  - The **life cycle** of a given object
  - The **events** causing a transition from one state to another
  - The **actions** that result from a state change
- State transition diagrams are created for objects with **significant dynamic behavior**
- Sequence diagrams are examined to define statechart diagram of a class

Statechart Diagram



Initial

login()

Login Dialog

do/query

cancel()

Failed

[declined]

enter(name,password)

Authentication

[accepted]

Success

Initial State

State

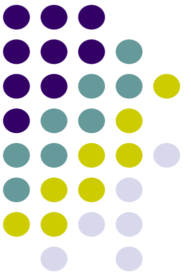
Transition enabled by the event

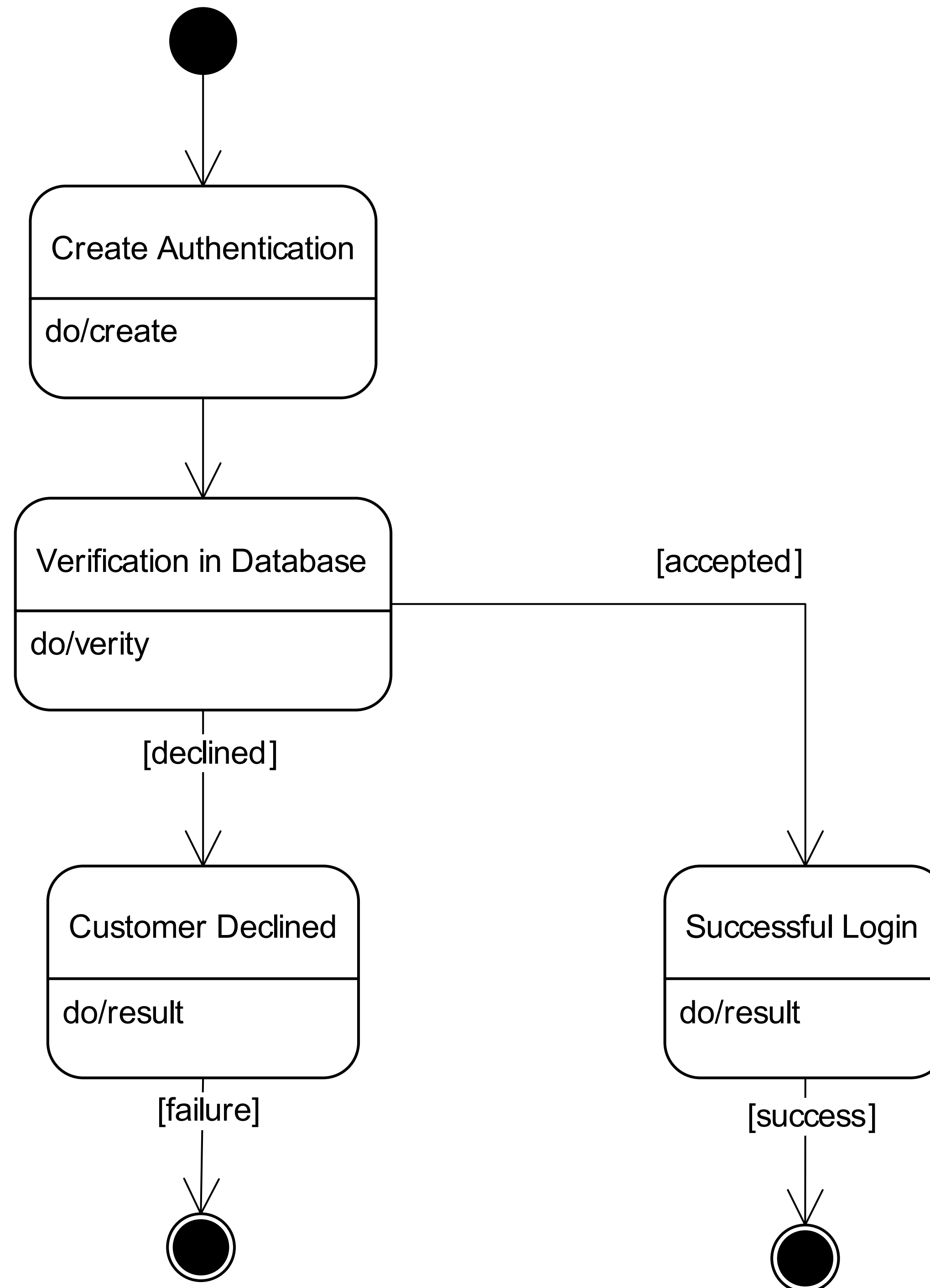
Composite State

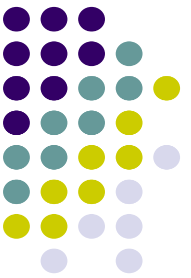
Final State

A **statechart diagram** for a class Login

Elaborated Composite State



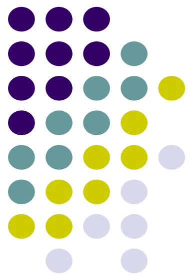




Workflow Modeling

- **Activity Diagram** is a variation of a state machine in which the states represent the performance of **activities** and the transitions are triggered by their **completion**.
  - The purpose of this diagram is to focus on flows driven by internal processing.

Activity Diagram



Customer
Accountant
Salesman

Issue an Order
Order
Accept an Order
Order Execution
Issue an Invoice
Invoice
Paying
Payment
Payment Acceptance
Shipping the Product
[unrealizable]
Case Closed

Control Flow

Object Flow

Action State (Activity)

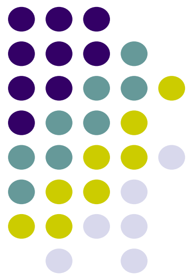
Decision

Fork Transition

Object

Join Transition

Actions may be organized into **swimlanes**. Swimlanes are a kind of package for organizing **responsibility for activities** provided by workers.



Object Collaboration

An collaboration diagram emphasizes the structural organization of objects that send and receive messages.

Synchronous message and its return value

1: register
2: query
3: enter(login, password)
3.3: result

3.2.3 *[i = 1..n]: verifyCustomer(i, login, password)

3.2: response = verify(aut)

«self»

: Login

«association»

db : CustomerDatabase

«global»

3.2.1: login = getLogin()
3.2.2: password = getPassword()

3.1: <<create>>

«local»

«parameter»

aut : Authorization

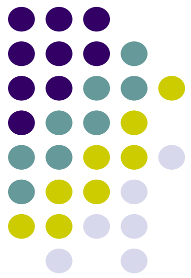
Object

Asynchronous message

Link between objects

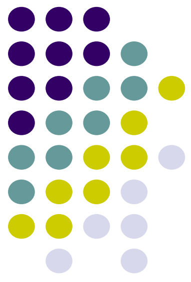
Visibility

Exercises



- What is the difference between sequence and collaboration diagram?
- What is the purpose of state chart diagram?
- Build a sequence diagram for the task of **money withdrawal from ATM**. Consider interaction between Client, ATM and Credit Card.

Implementation View of the System

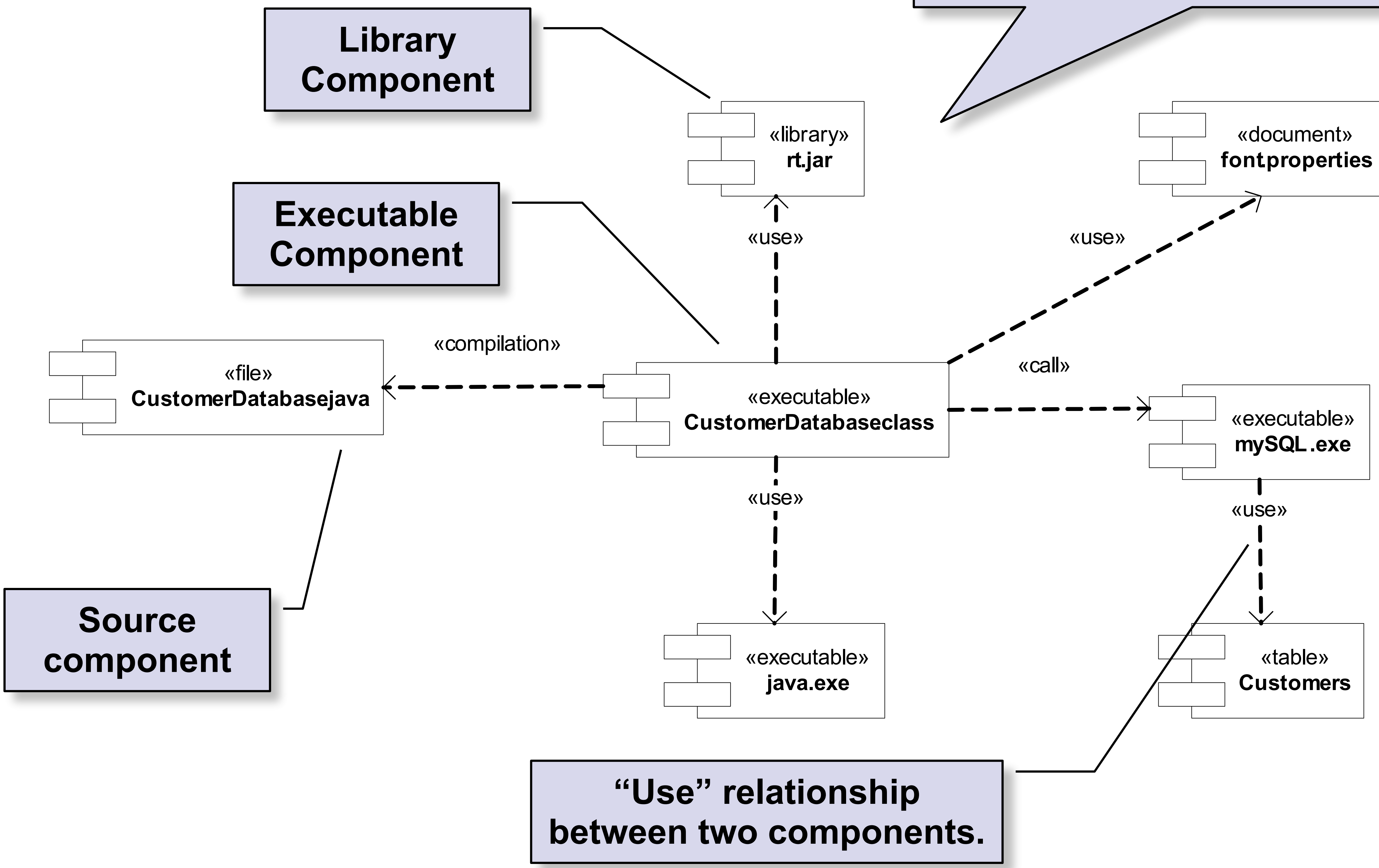


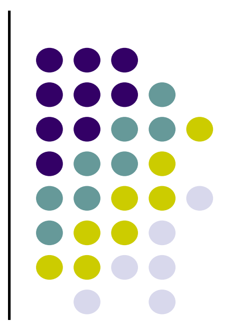
The goal of the implementation workflow is to flesh out the designed architecture and the system as a whole.

- **Implementation Model** describes how elements in the design model, such as design classes, are implemented in terms of components such as a source code files, executables, and so on. The implementation model also describes **how the components are organized** according to the structuring and modularization mechanism available in the implementation environment and the programming language (e.g. *package* in Java).

Component Diagram

Component Diagram illustrates the organization and dependencies among software components.





Deployment Diagram

e-Shop

«library»
rt.jar

«document»
font.properties

«use»

«use»

«executable»
CustomerDatabaseclass

«call»

«use»

«executable»
java.exe

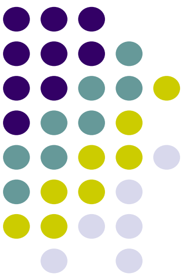
Database Server

«executable»
mySQL.exe

«use»

«table»
Customers

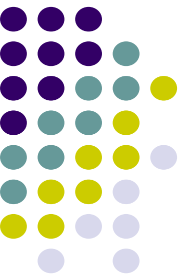
Deployment diagram shows the configuration of run-time processing elements.



Exercises

- Define deployment diagram for e-Shop based on .Net technology.

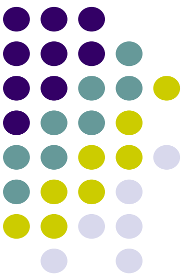
Formal Methods



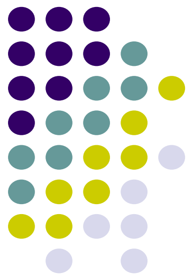
Techniques for the precise and unambiguous specification of software

- Formal methods include
  - Formal specification
  - Specification analysis and proof
  - Transformational development
  - Program verification
- Language for formal specification has to have precise and unambiguous syntax and semantics.

Mathematical Representation of Software

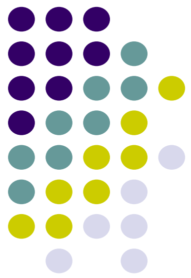


- Formal specifications are expressed in a mathematical notation with precisely defined vocabulary, syntax and semantics.
- Algebraic approach
  - The system is specified in terms of its operations and their relationships.
- Model-based approach
  - The system is specified in terms of a state model that is constructed using mathematical constructs such as sets and sequences.



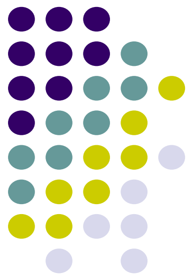
Finite Automata

- Finite automata are defined by
  - Q – finite set of states
  - I – finite set of inputs
  - $\delta: Q \times I \rightarrow Q$ – state transition function
- Visualization is based on statechart diagrams

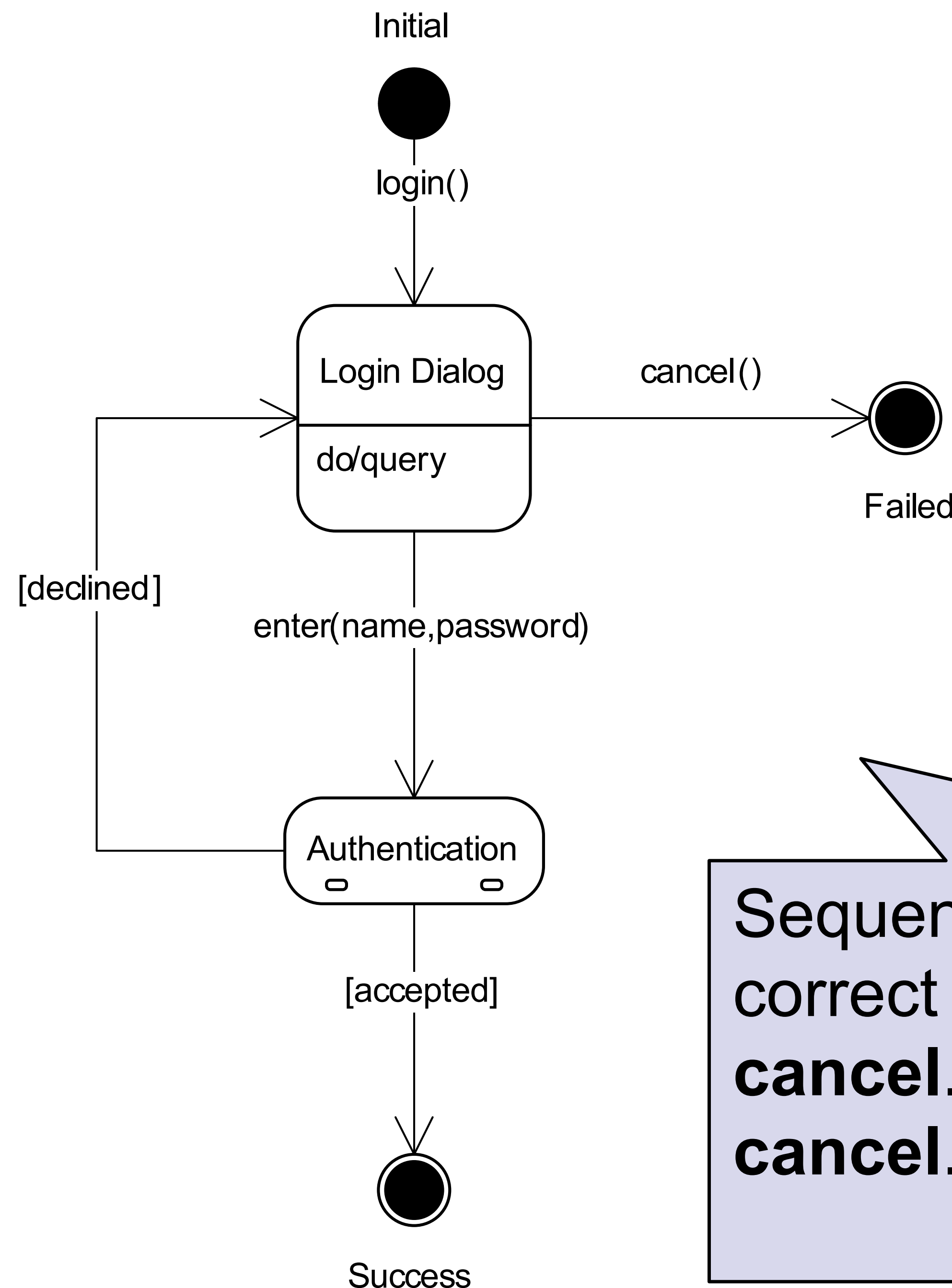


Finite Automata Cont …

- Initial and final states
  - q_0 - initial state
  - $F \subseteq Q$ - set of final states
- Sequence of inputs is accepted by automaton when a final state is reached from the initial state.



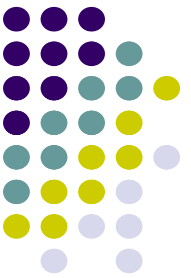
Example: Login behaviour



- Q = {*Initial, Login Dialog, Authentication, Success, Failed*}
- I = {*login, enter, cancel, accepted, declined*}
- δ_1(*Initial, login*) = *Login Dialog*
  δ_2(*Login Dialog, cancel*) = *Failed*
  δ_3(*Login Dialog, enter*) = *Authentication*
  δ_4(*Authentication, accepted*) = *Success*
  δ_5(*Authentication, declined*) = *Login Dialog*
- q_0 = *Initial*
- F = {*Success, Failed*}

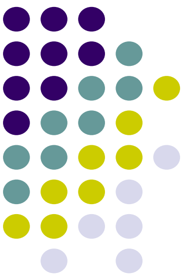
Sequence of inputs **login, enter, accepted** is correct as well as **login, enter, declined, cancel**. Not acceptable is **login, enter, cancel**.

Key Issues



- Formal methods reduce number of errors in software but the mathematical representation requires more time to market software product.
- Formal methods are hard to scale up to large systems.
- The main area of their applicability is critical systems. In this area the use of formal methods seems to be cost-effective.
- The idea is to combine formal methods with diagrammatic languages like UML. The formal methods specify what cannot be captured by diagrams.

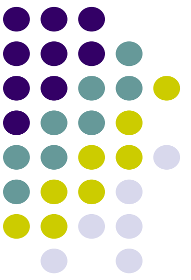
MDA – Model Driven Architecture



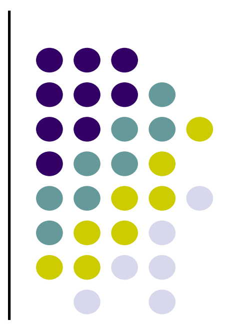
MDA is a framework that defines how models defined in one language can be transformed into models in other languages.

- The software development process is driven by the activity of modeling of software system.
- The MDA process is divided into three steps:
  - Build a model with high level of abstraction that is independent of any implementation technology (Platform Independent Model – PIM)
  - Transform the PIM into one or more model tailored for implementation constructs specific to actual implemenation technology (Platform Specific Model – PSM)
  - Transform the PSMs to code.
- Key requirement: automation of transformations. Models have to be precise and unambiguous.

UML Limitations



- UML modeling is based on producing diagrams. The information conveyed by such a model has a tendency to be incomplete, imprecise and even inconsistent.
- Diagrams cannot express the statements that should be part of a specification.
- Interpretation of model can be unambiguous.



UML Limitations: Example

Product
-name : string
-price : float
+discount(in percentage : int) : float

-products
3 orders
-buyer
0..*
0..1

Customer
-name : string
-address : string
-e-mail : string
+pay(in amount : float)

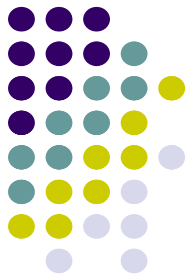
1..*
-items
-payer
1

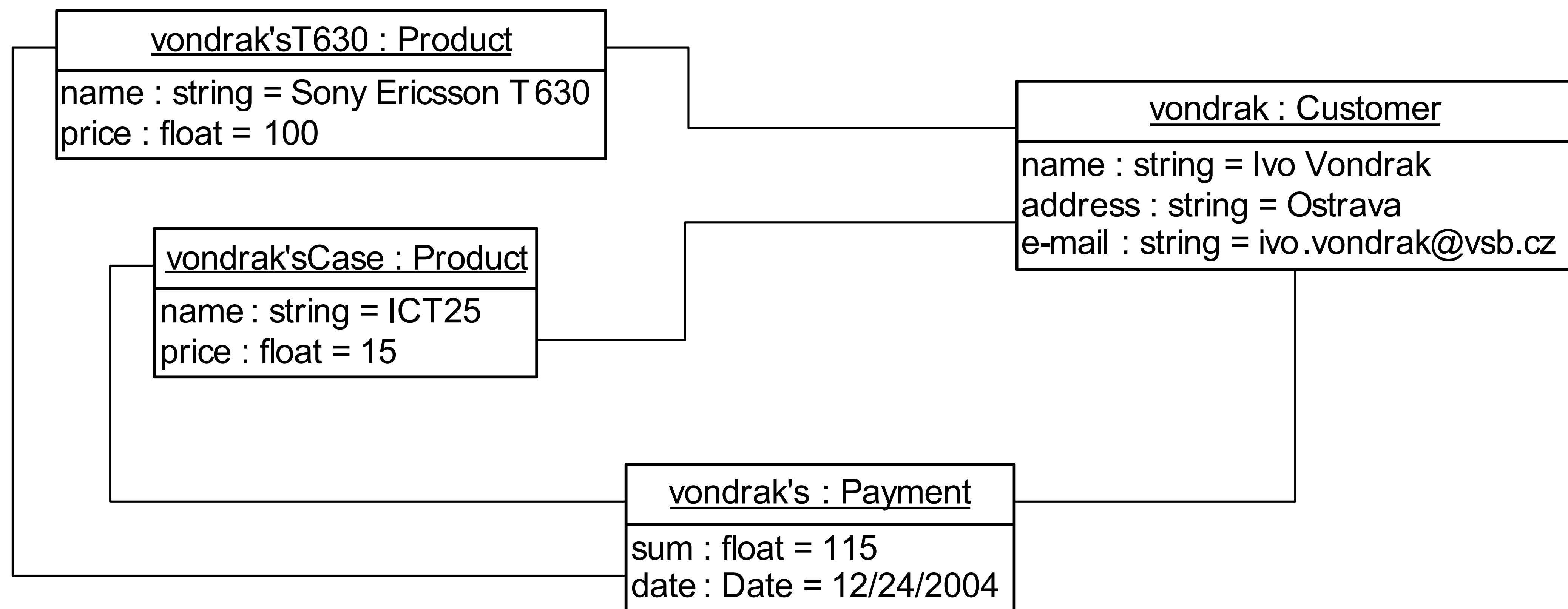
defines4
-payment

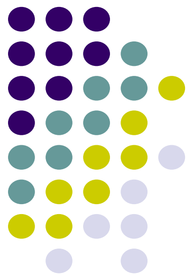
Payment
-sum : float
-date : Date

-remittance
3 pays
0..1
0..1

Correct Intepretation of Model







Mistaken Interpretation

vondrak'sT630 : Product
name : string = Sony Ericsson T630 price : float = 100

vondrak'sCase : Product
name : string = ICT25 price : float = 15

smith'sP900 : Product
name : string = Sony Ericsson P900 price : float = 200

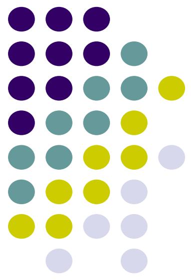
vondrak : Customer
name : string = Ivo Vondrak address : string = Ostrava e-mail : string = ivo.vondrak@vsb.cz

smith : Customer
name : string = John Smith address : string = Los Angeles e-mail : string = john.smith@matrix.com

vondrak's : Payment
sum : float = 200 date : Date = 12/24/2003

Object **vondrak** should pay his products and the amount that has to be paid is given by the sum of both product. This intepreation is not valid but it is correct from point of view of class diagram. **Class diagram is unambiguous!**

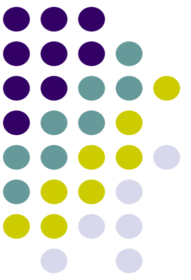
OCL – Object Constraint Language



OCL is a language that can express additional and neccessary information about models and other artifacts used in object-oriented modeling, and should be used in conjuction with UML diagrammatic models.

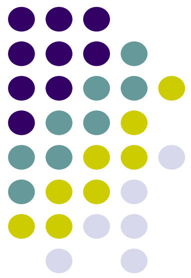
- OCL is precise, unambiguous language that is easy for people who are not mathematicians or computer scientists to understand. It does not use any mathematical symbols, while maintaining rigor in its definition.
- OCL is a typed language, because it must be possible to check an expression included in a specification without having to produce an executable version of the model.
- OCL is a declarative, side-effects-free language; that is, the state of a system does not change because of an OCL expression.
- UML/OCL enables to build models completely platform-independent.

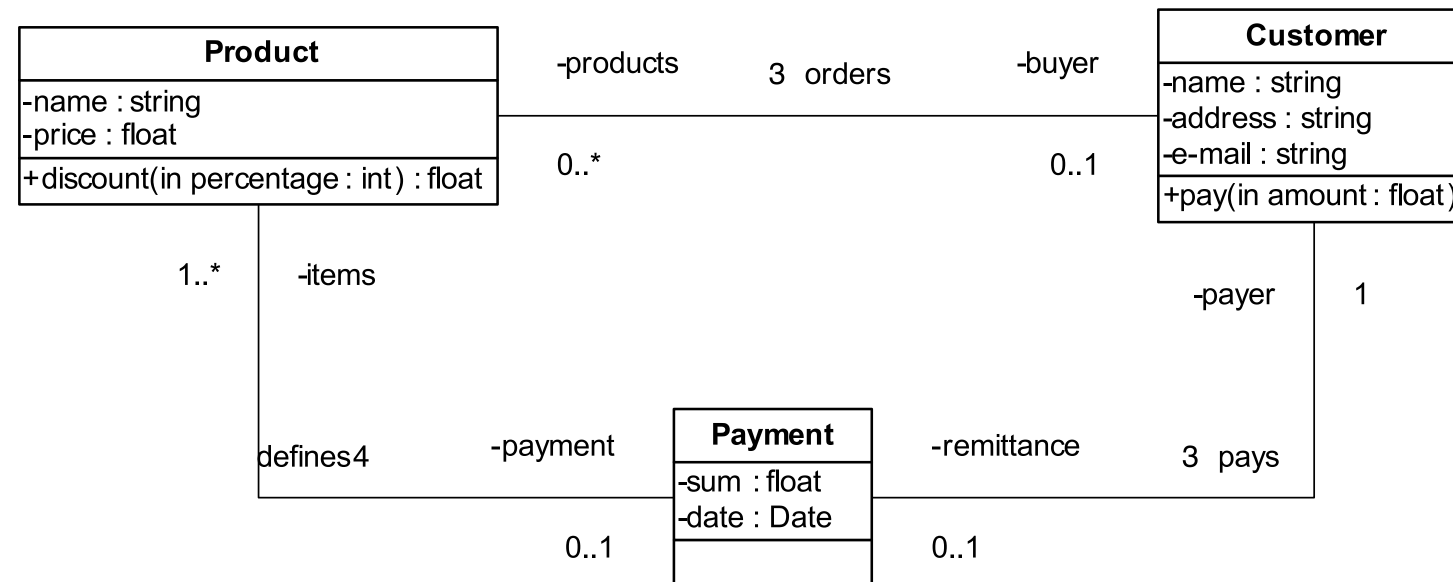
How to Combine UML and OCL



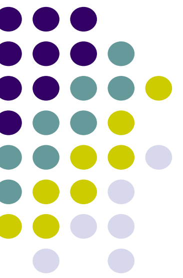
- Rules related to the class diagram of e-shop:
  - A payment is valid only when the products ordered by a customer are paid by the same customer.
  - The e-mail address has to be unique.
  - The payment sum is given by the sum of all ordered products' prices.
  - Customer has to pay amount of money equal to the payment sum.

OCL Expressions





- ***context*** *Payment*
  inv: *self.items->forAll (item | item.buyer = payer)*
- ***context*** *Customer*
  inv: *Customer::allInstances()->isUnique(e-mail)*
- ***context*** *Payment*
  inv: *self.items.price->sum() = self.sum*
- ***context*** *Customer::pay(amount: float)*
  pre: *self.remittance.sum = amount*



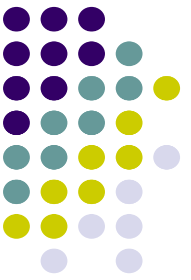
Invariants

An invariant is a constraint that should be true for object during its complete lifetime.

- Invariants on attributes
  context *Product*
  inv *ofPrice: price >= 0*

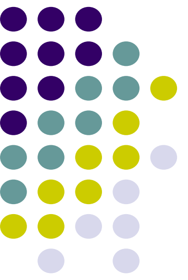
- Invariants on associated objects
  context *Payment*
  inv *ofAge: payer.age >= 18*

Working with Collections of Objects



- When the multiplicity of an association is greater than 1, OCL provides wide range of collection operations:
  - The **size** operation
    context *Payment*
    inv *minItems: items->size() >= 1*
  - The **forAll** operation
    context *Payment*
    inv *noPayment: items.forAll(price = 0)* ***implies*** *sum = 0*
  - Other oprations: **select**, **isEmpty**, **collect** …

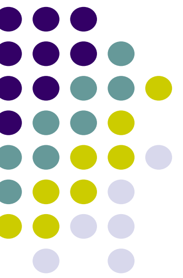
Preconditions and Postconditions



Preconditions and postconditions are constraints that specify the applicability and effect of an operation without stating an algorithm or implementation.

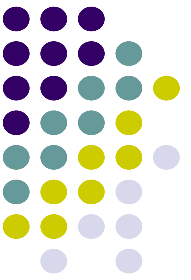
```
context Product::discount(percentage: Integer) : Real
  pre: percentage >= 0 and percentage <= 10
  post:
        let oldPrice : Real = self.price@pre
        in self.price = oldPrice – (oldPrice*percentage/100)
        and result = self.price
```

Exerices

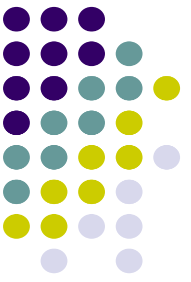


- What is definition of Invariant?
- **Specify the invariant for class Customer that reflects following constraint:** Customer’s payment contains all products that were ordered by the customer.

Design Patterns

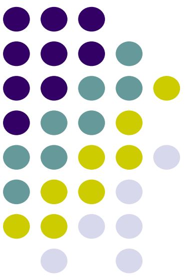


- The design pattern concept can be viewed as an **abstraction of imitating useful parts** of other software products.
- The design pattern is **description of communicating objects and classes** that are customized to solve a general design problem in a particular context.



Classification of Design Patterns

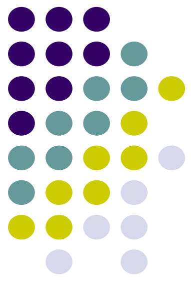
- **Creational patterns** defer some part of object creation to a subclass or another object.
- **Structural patterns** composes classes or objects.
- **Behavioral patterns** describe algorithms or cooperation of objects.



Creational Design Patterns

- **Factory Method** define an interface for creating an object, but let subclasses decide which class to instantiate.
- **Factory** provides an interface for creating families of related objects without specifying their concrete classes.
- **Prototype** specifies the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Factory Method Example



Document d = self.createDocument()

Application
+newDocument() +createDocument() : Document

Document
+open() +close() +save()

1

*

WordApplication
+createDocument() : Document

PDFApplication
+createDocument() : Document

PDFDocument

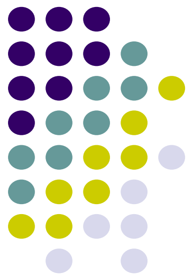
WordDocument

«instantiate»

«instantiate»

return new WordDocument

Factory Method define an interface for creating an object, but let subclasses decide which class to instantiate.



Factory Method Design Pattern

...
p = factoryMethod()
...

Creator

+factoryMethod()
+anOperation()

Product

ConcreteCreator

+factoryMethod() : Product

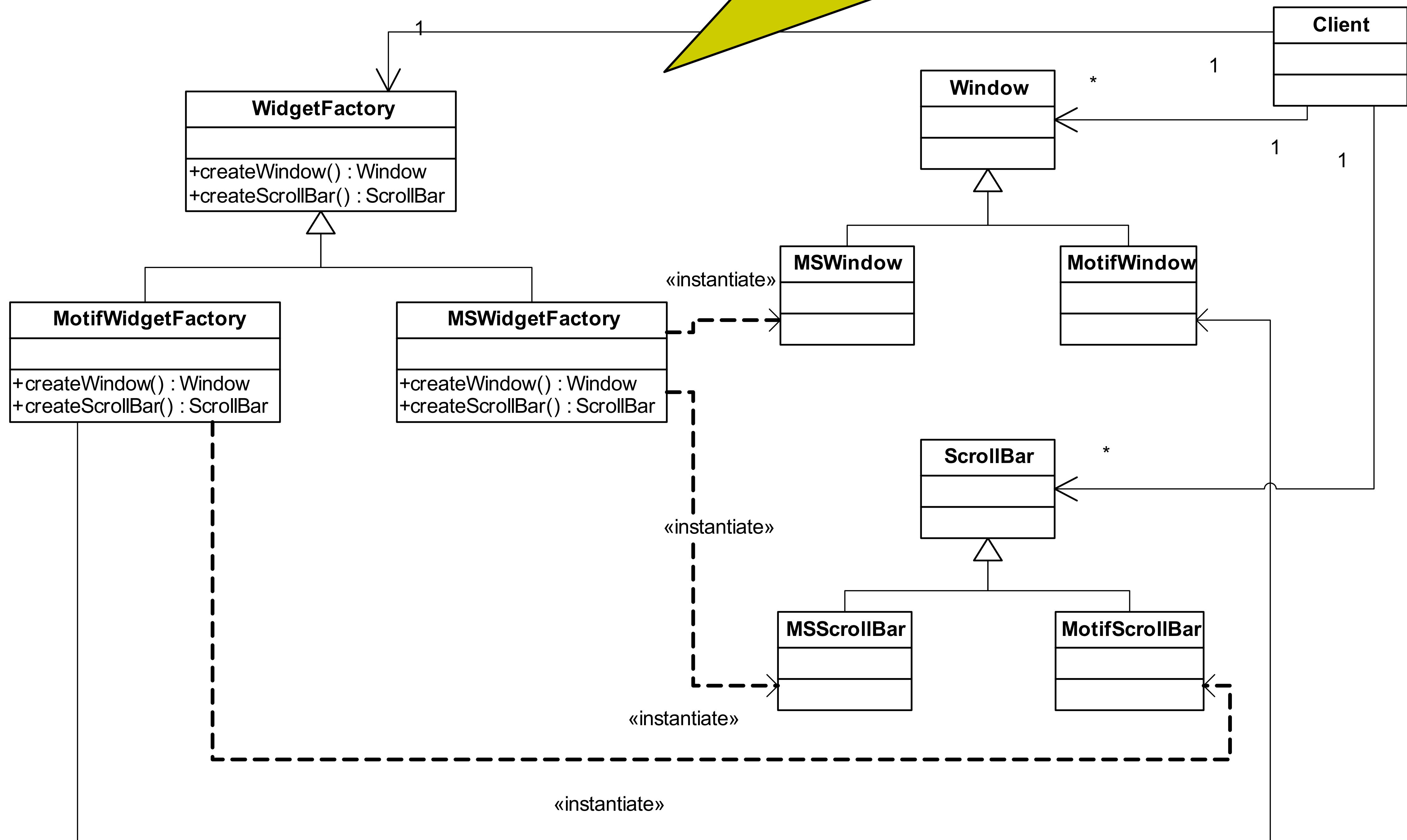
«instantiate»

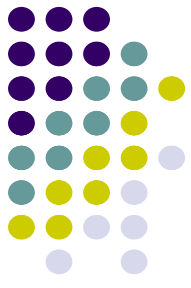
ConcreteProduct

return new ConcreteProduct()

Factory Example

Factory provides an interface for creating families of related objects without specifying their concrete classes.





Factory Design Pattern

Client

1

AbstractFactory

+createProductA() : AbstractProductA
+createProductB() : AbstractProductB

AbstractProductA

*

1

1

1

ProductA1

ProductA2

«instantiate»

ConcreteFactory2

+createProductA() : AbstractProductA
+createProductB() : AbstractProductB

ConcreteFactory1

+createProductA() : AbstractProductA
+createProductB() : AbstractProductB

AbstractProductB

*

«instantiate»

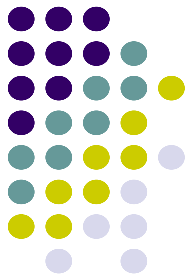
ProductB1

ProductB2

«instantiate»

«instantiate»

Prototype Example



Tool
+insert()

-prototype

1   1

Graphic
+draw(in position) +clone() : Graphic

Rectangle
+draw(in position) +clone() : Graphic

Circle
+draw(in position) +clone() : Graphic

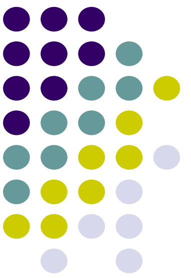
```
g = prototype.clone()
while (user drags mouse) {
    g.draw(new position);
}
insert symbol into drawing;
```

return copy of self

return copy of self

Prototype specifies the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Prototype Design Pattern



Client

+operace()

-prototype

1

1

Prototype

+clone() : Prototype

p = prototype.clone()

ConcretePrototype1

+clone() : Prototype

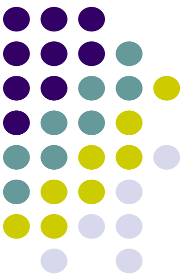
ConcretePrototype2

+clone() : Prototype

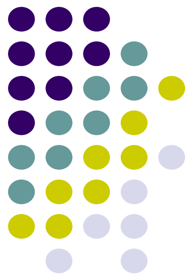
return copy of self

return copy of self

Structural Design Patterns



- **Composite** composes objects into tree structures to represent part-whole hierarchies. Composite lets client treat individual objects and compositions of objects uniformly.
- **Adapter** converts the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- **Decorator** attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- **Proxy** provides a surrogate or representative for another object to control access to it.



Composite Example

Graphic

+add(in g : Graphic)
+remove(in g : Graphic)
+getChild(in index : int) : Graphic
+draw()

Client

1

1

*

-component

Rectangle

+draw()

Circle

+draw()

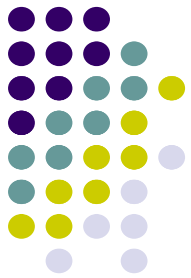
Drawing

+add(in g : Graphic)
+remove(in g : Graphic)
+getChild(in index : int) : Graphic
+draw()

1

for (all graphics in picture)
component.draw()

Composite composes objects into tree structures to represent part-whole hierarchies. Composite lets client treat individual objects and compositions of objects uniformly.



Composite Design Pattern

Client

1

1

Component

+add(in c : Component)
+remove(in c : Component)
+getChild(in index : int) : Component
+operation()

*

-component

Leaf

+operation()

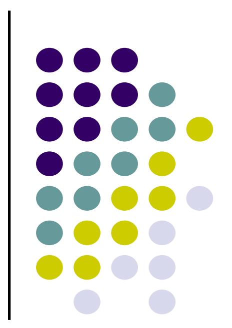
Composite

+add(in g : Component)
+remove(in g : Component)
+getChild(in index : int) : Component
+operation()

1

for (all components)
    component.operation()

Adapter Example



DrawingEditor

Shape
+getBoundingBox() : Rectangle

1 *

LineShape
+getBoundingBox() : Rectangle

TextShape
+getBoundingBox() : Rectangle

-text

TextView
+getExtent() : Rectangle

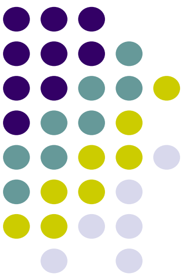
1

1

return text.getExtent()

Adapter converts the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Adapter Design Pattern



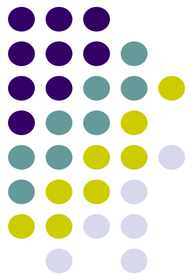
Client

Target
+request()

Adapter
+request()

Adaptee
+specificRequest()

Decorator Example



VisualComponent

+draw()

1

-component

TextView

+draw()

Decorator

-

+draw()

1

component.draw()

aBorderDecorator

component

aScrollDecorator

component

aTextView

BorderDecorator

+draw()
+drawBorder()

ScrollDecorator

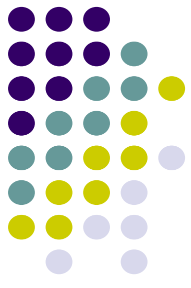
+draw()
+scrollTo()

super.draw()
drawBorder()

super.draw()
scrollTo()

Decorator attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Decorator Design Pattern



Component

+operation()

1

-component

ConcreteComponent

+operation()

Decorator

-

+operation()

1

component.operation()

ConcreteDecorator1

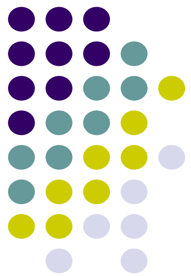
+operation()
+addedOperation1()

ConcreteDecorator2

+operation()
+addedOperation2()

super.operation()
addedOperation2()

Proxy Example



DocumentEditor

Element
+draw()
+getExtent()

1 *

Text
-extent
-content
+draw()
+getExtent()

Image
-imageImp
+draw()
+getExtent()

-image
1 1

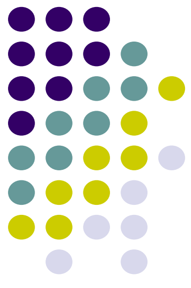
ImageProxy
-fileName
-extent
+draw()
+getExtent()
+load()

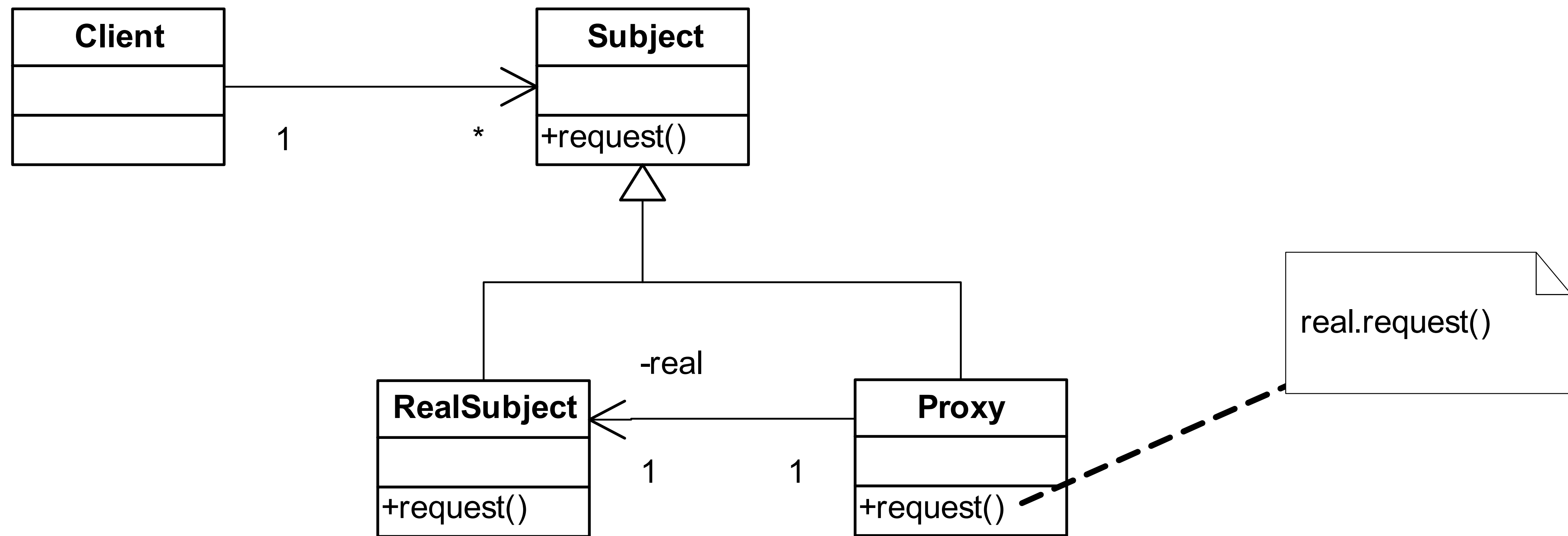
```
if (image == null)
   image = load(fileName)
image.draw()
```

```
if (image == null)
   return extent
else
   return image.getExtent()
```

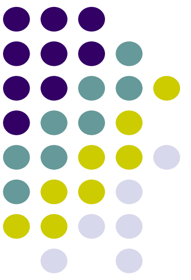
Proxy provides a surrogate or representative for another object to control access to it.

Proxy Design Pattern

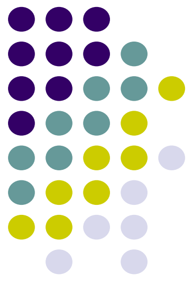




Behavioral Design Patterns



- **Chain of Responsibility** avoids coupling the senders of a request to its receiver by giving more than one object a chance to handle request. Chain the receiving objects and pass the request along the chain until an object handles it.
- **Command** encapsulates a request as an object, thereby letting you parametrize clients with different requests, queue or log requests, and support undoable operations.
- **Iterator** provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- **Observer** defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **State** allows an object to alter its behaviour when its internal state changes. The object will appear to change its class.
- **Strategy** defines a family of algorithms, encapsulates each one, and makes them interchangable. Strategy lets algorithm vary independently from clients that use it.



Chain of Responsibility Example

1 -handler

Client

1 1

HelpHandler

+handleHelp()

handler.handleHelp()

Application

+handleHelp()
+showHelp()

Widget

aClient

handler

aButton

handler

aDialog

handler

anApplication

handler

Dialog

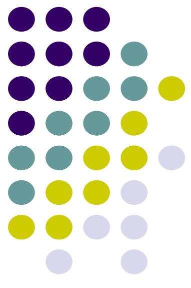
+handleHelp()
+showHelp()

Button

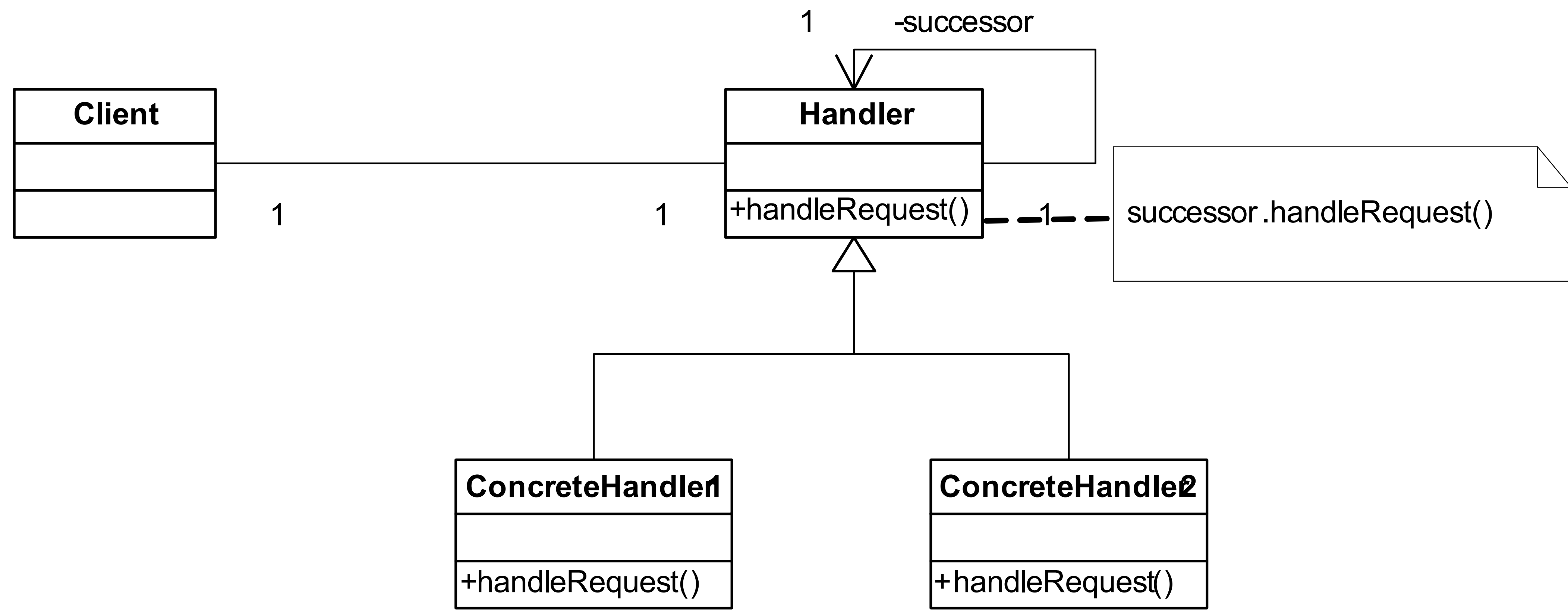
+handleHelp()
+showHelp()

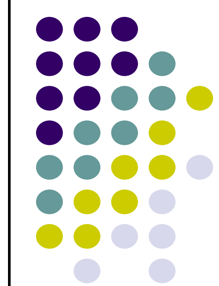
if (can handle)
showHelp()
else
super.handleHelp()

Chain of Responsibility avoids coupling the senders of a request to its receiver by giving more than one object a chance to handle request. Chain the receiving objects and pass the request along the chain until an object handles it.



Chain of Responsibility Design Pattern





Command Example

Application
+add(in doc : Document)

Menu
+add(in m : MenuItem)

MenuItem
+clicked()

-command

Command
+execute()

1 * 1 * 1 1

1

command.execute()

-application 1

Document
+open()
+close()
+cut()
+copy()
+paste()

-document

PasteCommand
+execute()

AddCommand
+execute()
+askUser() : string

* 1 1

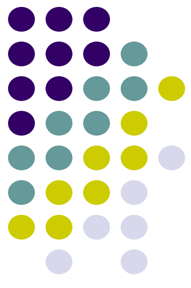
document.paste()

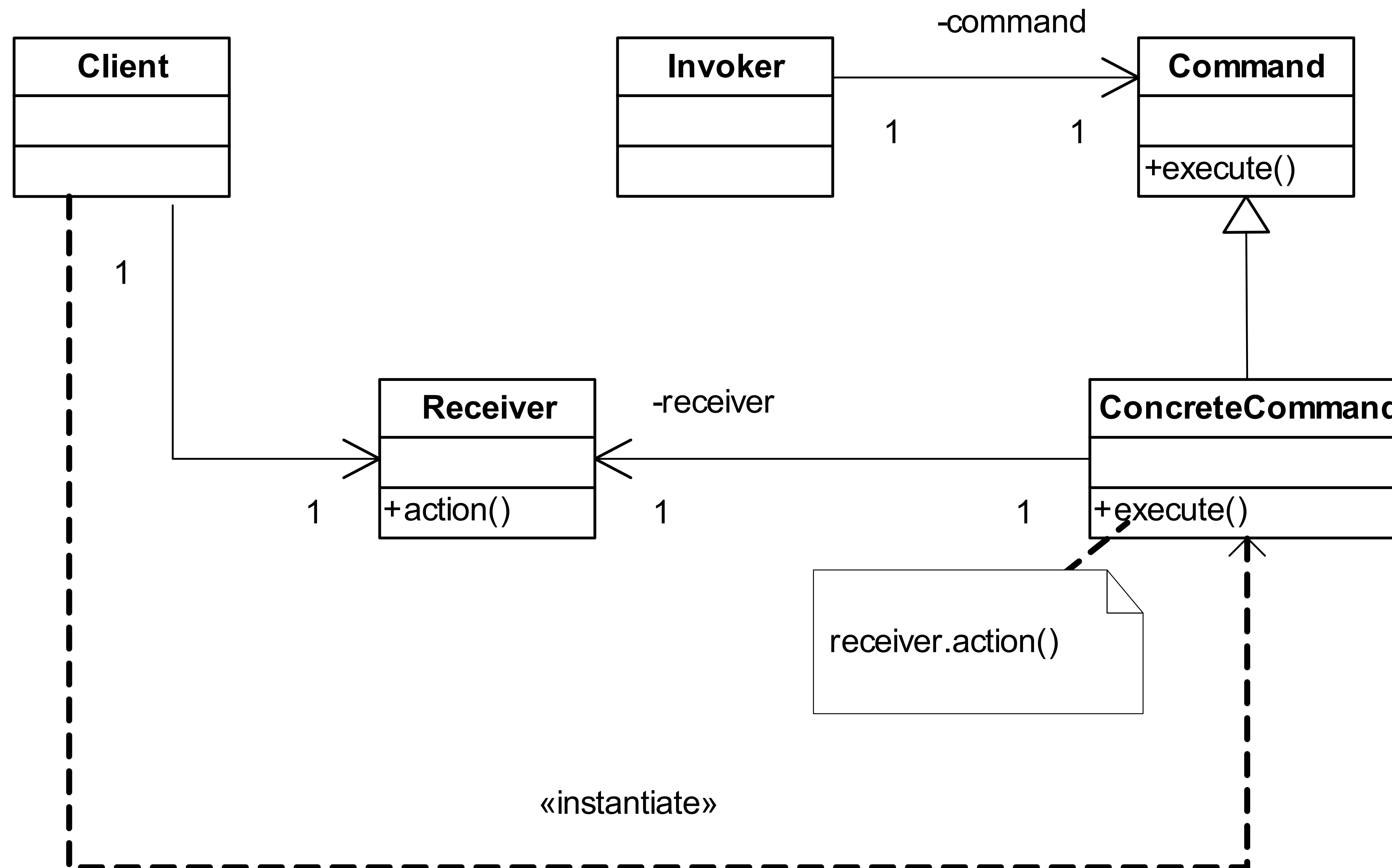
1

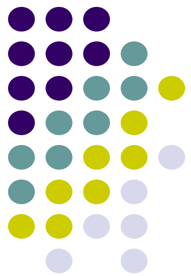
name = askUser()
doc = new Document(name)
application.add(doc)
doc.open()

Command encapsulates a request as an object, thereby letting you parametrize clients with different requests, queue or log requests, and support undoable operations.

Command Desing Pattern







Iterator Example

Collection
+createIterator() : Iterator

Client

Iterator
+first()
+next()
+isDone()

1 1 1 1

List
+createIterator() : Iterator
+append()
+remove()
+get(in index)

Stack
+createIterator() : Iterator
+pop()
+push()
+top()

«instantiate»

StackIterator
+first()
+next()
+isDone()

ListIterator
+first()
+next()
+isDone()

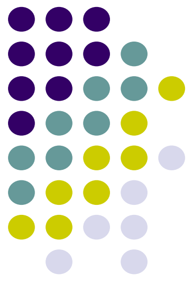
1 1

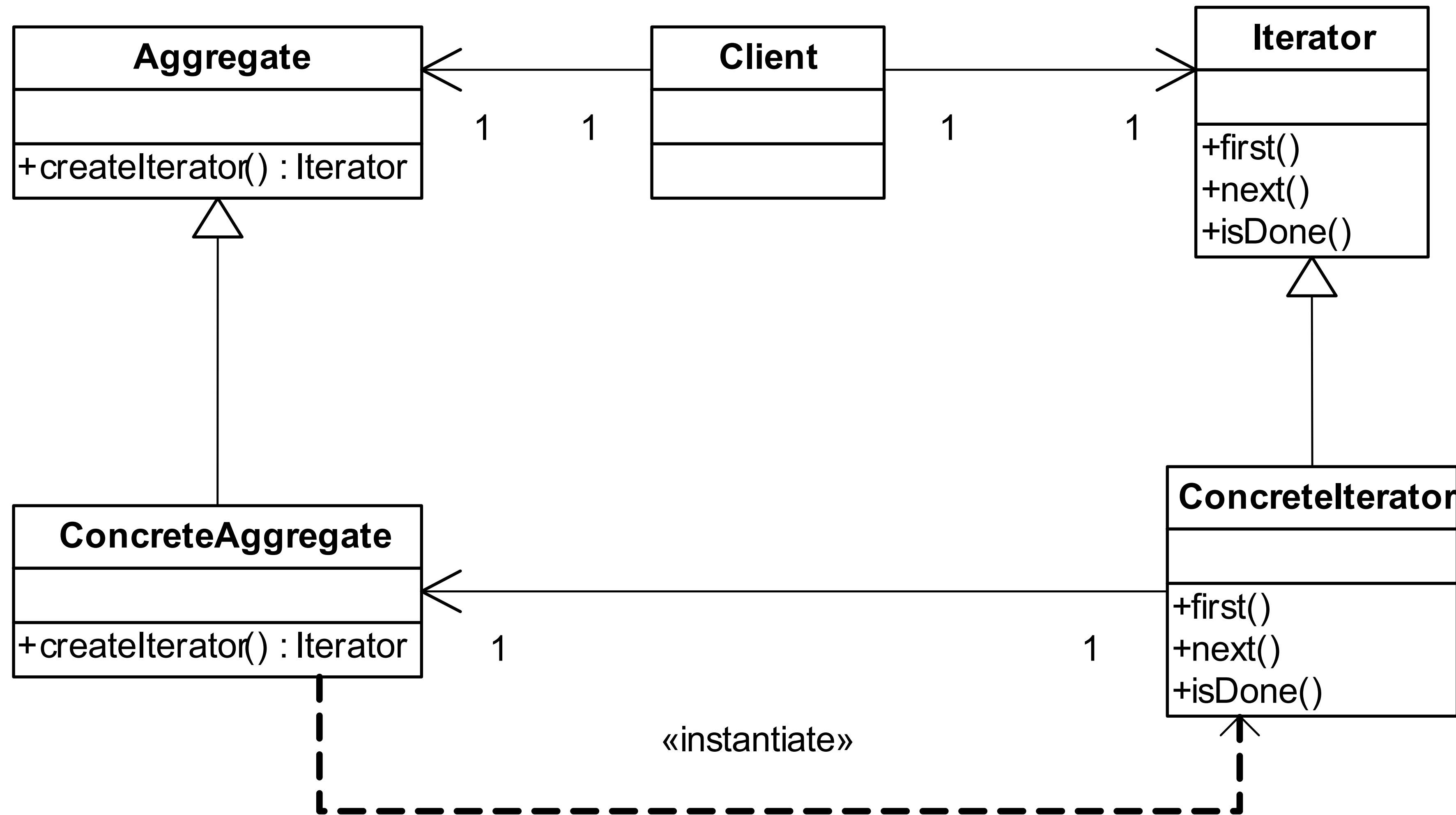
1 1

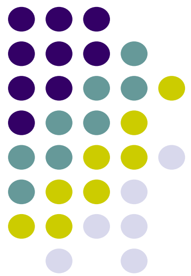
«instantiate»

Iterator provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Iterator Design Pattern







Observer Example

1 * -observer

Subject

+attach(in o : Observer)
+detach(in o : Observer)
+notify()

-subject

1 1

Observer

+update()

for all observers
observer.update()

StockMarket

-data[] : float

+getData() : float

TextArea

+update()
+print(in data[])

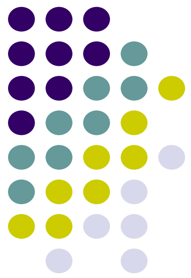
BarChart

+update()
+draw(in data[])

data = subject.getData()
print(data)

data = subject.getData()
draw(data)

Observer defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



Observer Design Pattern

1
*
-observer

Subject

+attach(in o : Observer)
+detach(in o : Observer)
+notify()

-subject

1
1

Observer

+update()

for all observers
observer.update()

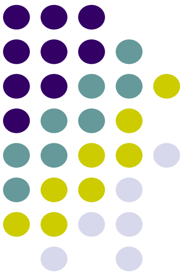
ConcreteSubject

+getState()

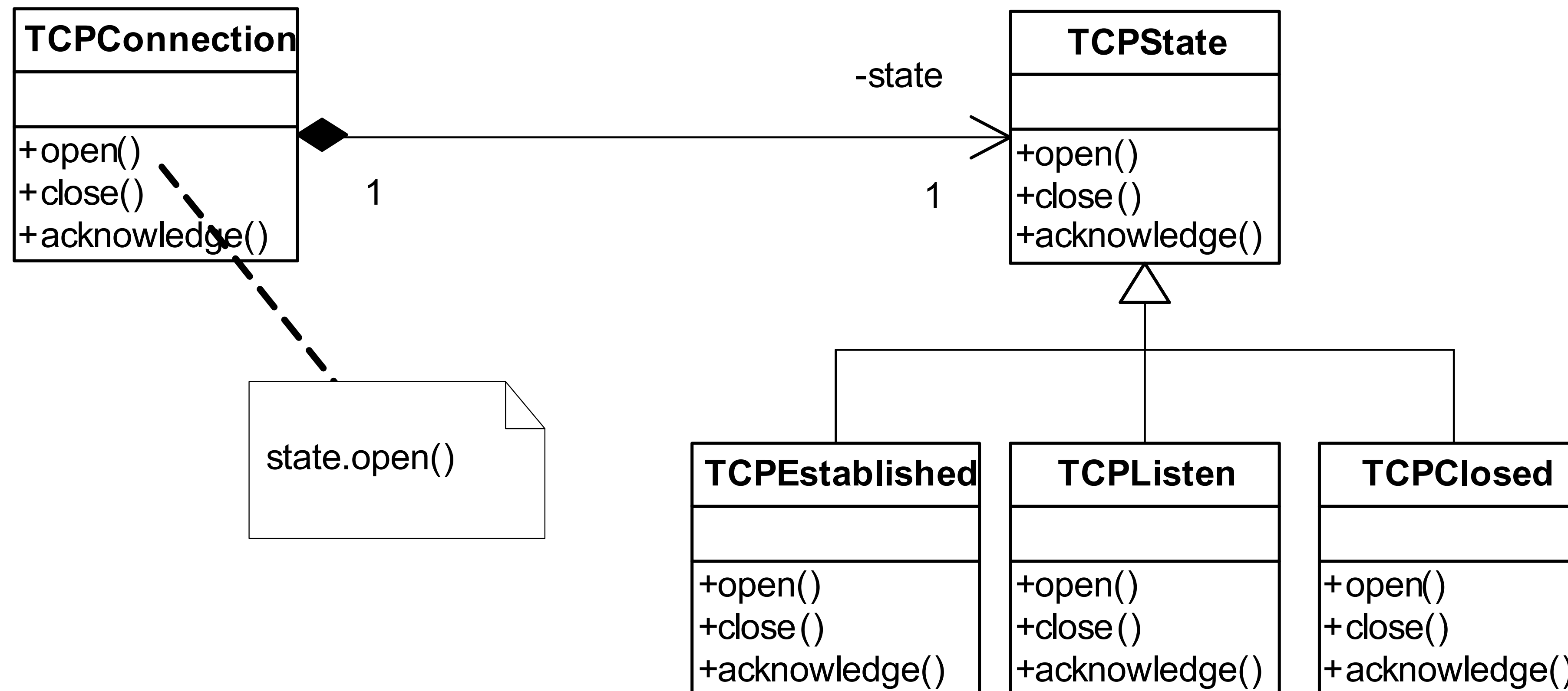
ConcreteObserver

+update()

subject.getState()

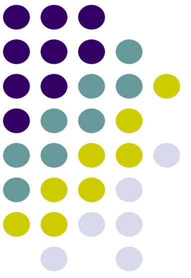


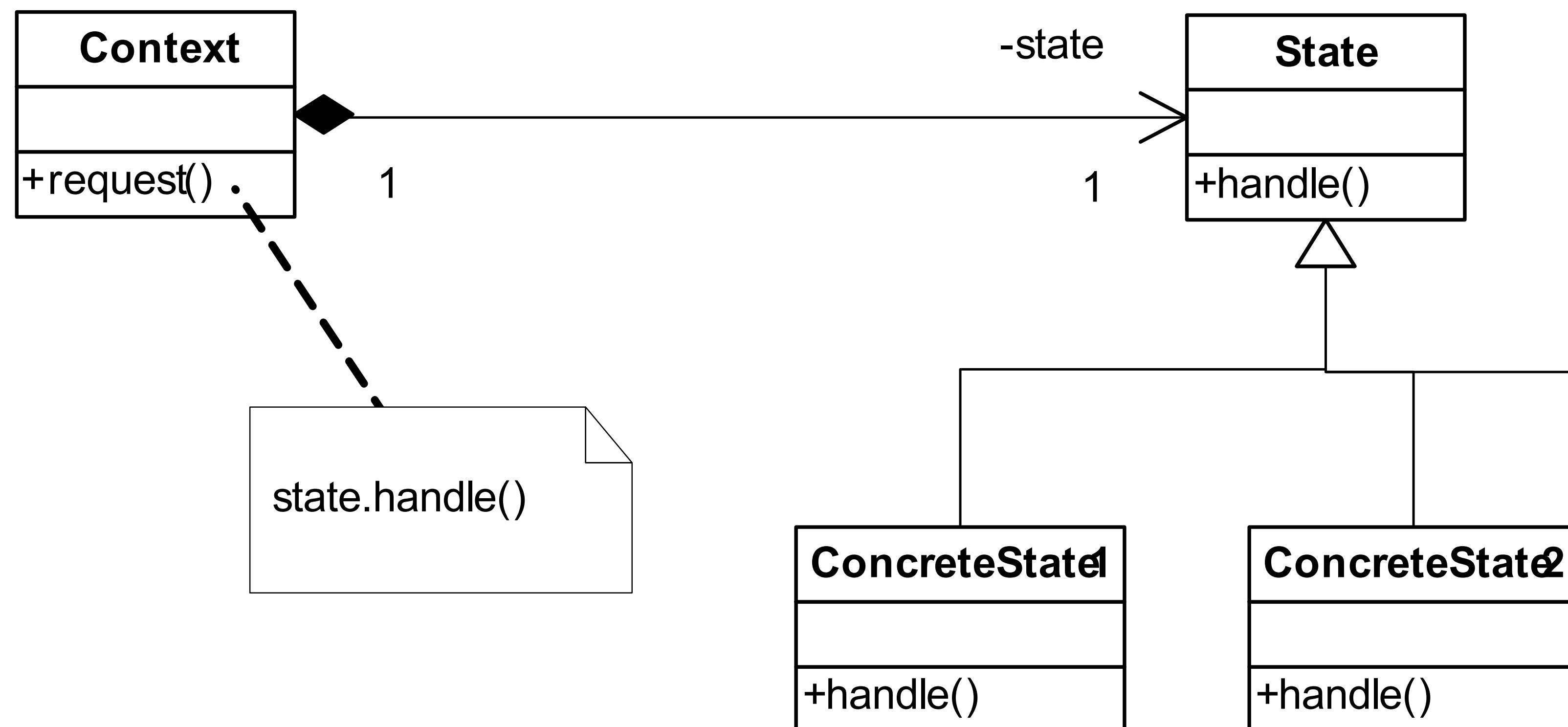
State Example



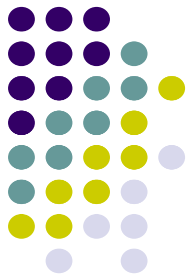
State allows an object to alter its behaviour when its internal state changes. The object will appear to change its class.

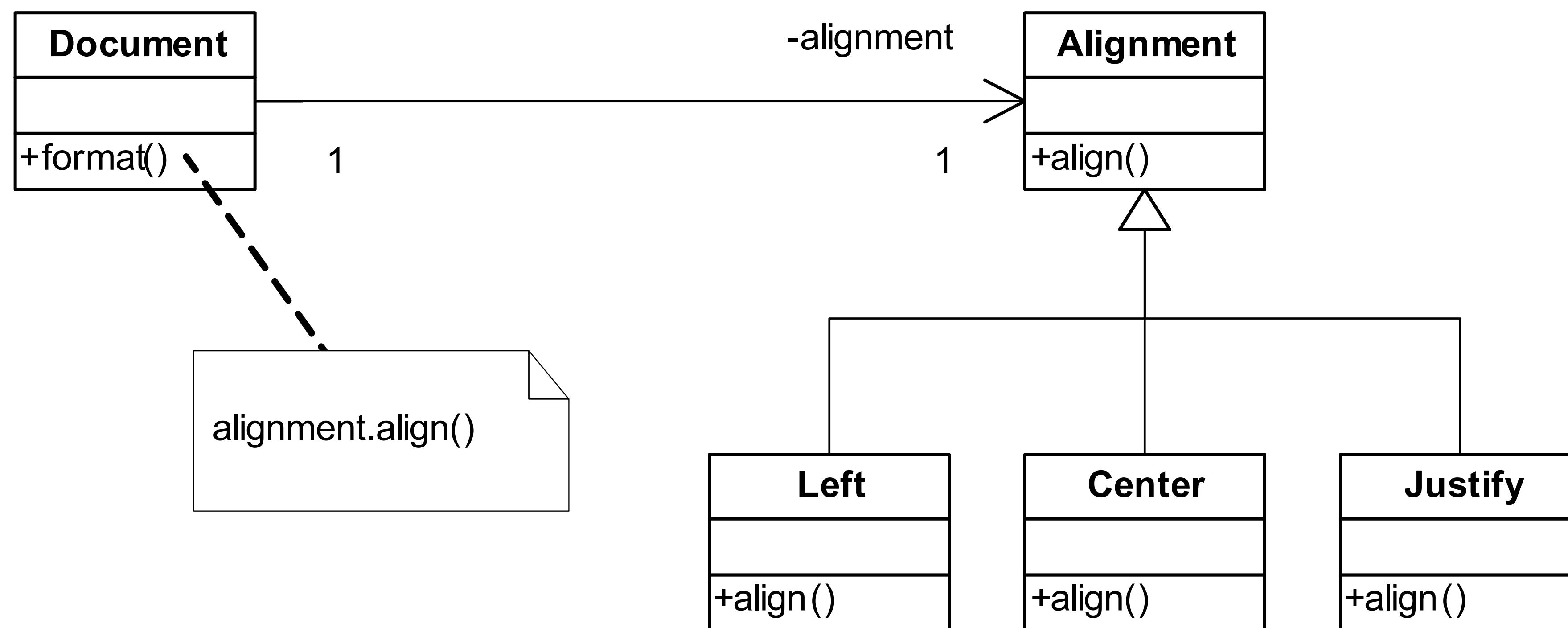
State Design Pattern





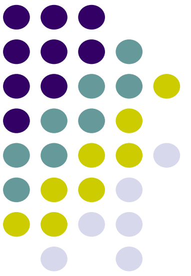
Strategy Example

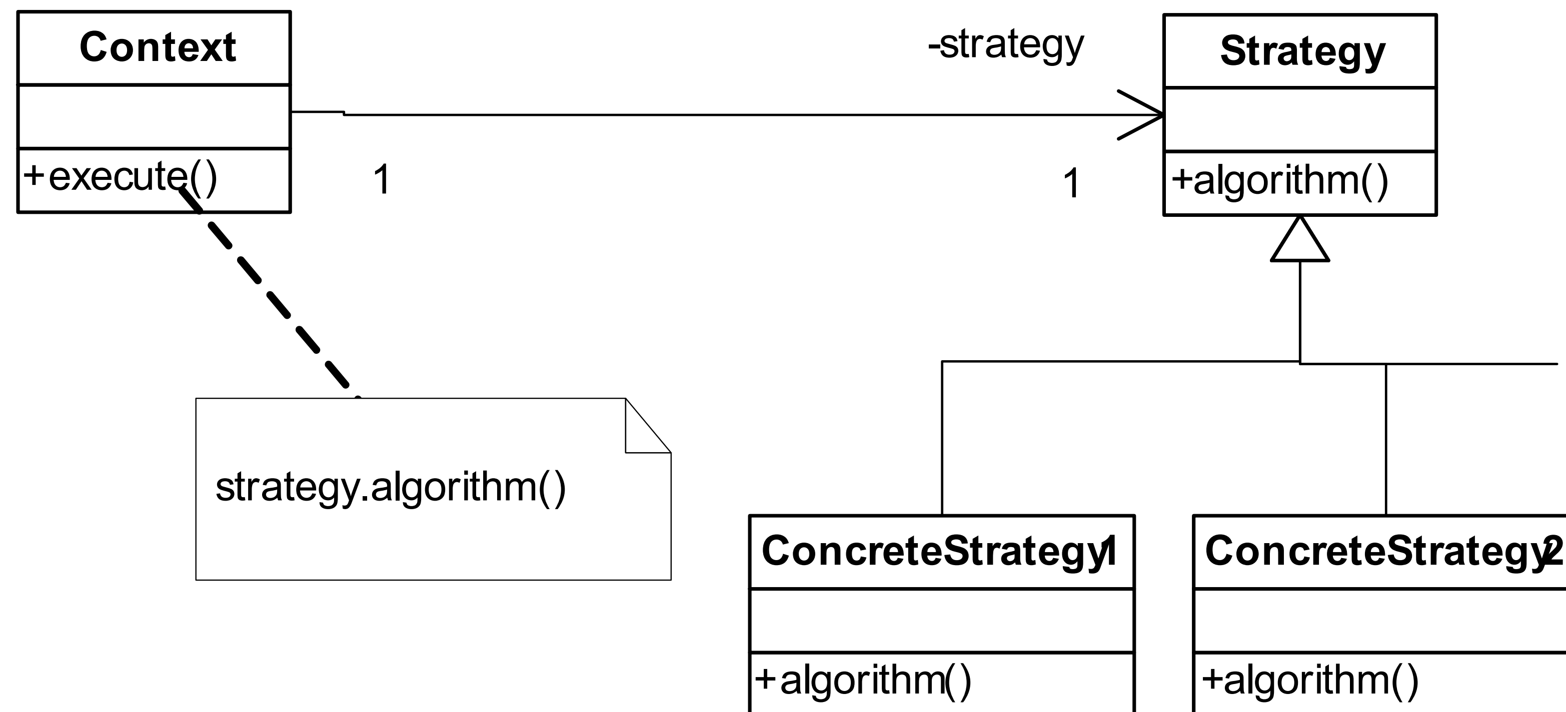




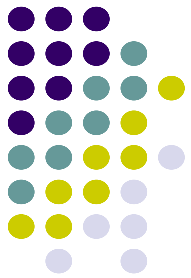
Strategy defines a family of algorithms, encapsulates each one, and makes them interchangable. Strategy lets algorithm vary independently from clients that use it.

Strategy Design Pattern





Exercises



- Let‘s assume that we have the class Application. **Use design pattern Prototype** to add new documents WordDocument and PDFDocument to the application.
- Specify simple file system that **employs design pattern Composite**.
- We have a class Table that consists of n instances of class Record. The class Recors defines three attributes name, age and sallary. The class Table defines operation sort that sorts records based on name, age or sallary. **Employ design pattern Strategy** for that purpose.